Computer verständlich

Rupert Röder

Einführung in Pascal

Garantiert Pascal lernen durch schrittweise Erarbeitung

FALKEN

Computer verständlich

Rupert Röder

Einführung in Pascal

Garantiert
Pascal lernen
durch
schrittweise
Erarbeitung

FALKEN

Titelhinweis:
Folgende Titel sind in der Reihe Computer verständlich bereits erschienen:
„Computer Grundwissen" (Nr. 4302)
„Falken Computer Lexikon" (Nr. 4185)
„Grundwissen Informationsverarbeitung" (Nr. 4314)
„Desktop Publishing: Typografie und Layout" (Nr. 4330)

CIP-Titelaufnahme der Deutschen Bibliothek

Röder, Rupert:
Einführung in Pascal : garantiert Pascal lernen
durch schrittweise Erarbeitung / Rupert Röder. –
Niedernhausen/Ts. : FALKEN, 1991
(Computer verständlich)
ISBN 3-8068-4325-2

ISBN 3 8068 4325 2

© 1990 by Falken-Verlag GmbH, 6272 Niedernhausen/Ts.
Die Verwertung der Texte und Bilder, auch auszugsweise, ist ohne Zustimmung des Verlags urheberrechtswidrig und strafbar. Dies gilt auch für Vervielfältigungen, Übersetzungen, Mikroverfilmung und für die Verarbeitung mit elektronischen Systemen.
Illustrationen: A. Janka, Offenbach / FALKEN Verlag

Die Ratschläge in diesem Buch sind von Autoren und Verlag sorgfältig erwogen und geprüft, dennoch kann eine Garantie nicht übernommen werden. Eine Haftung des Autors oder des Verlages und seiner Beauftragten für Personen-, Sach und Vermögenschäden ist ausgeschlossen.
Druck: Auer, Donauwörth

817 2635 4453 6271

Inhalt

9 **Einleitung**

11 **1 Grundlagen**
11 Maschinensprache
12 Computercodes
13 Pascal
14 Die Programmierumgebung
17 *Das Turbo-Hauptmenü*
18 *Die Ebenen der Computernutzung*
19 *Wordstar-Kompatibilität*
21 Aufgaben

22 **2 Das erste Programm**
24 Ein neuer Ansatz
25 Das erste Pascal-Programm
27 Programmerklärung und Regeln
30 Aufgaben

31 **3 Prozeduren und Konstanten**
32 Prozeduren
34 Lesen eines zusammengesetzten Pascal-Programms
35 Steueranweisungen für die Bildschirmausgabe
37 Feste Größen erhalten Namen
39 Bottom-up-Verfahren
40 Top-down-Verfahren
42 Konstanten für Zeichen und Textstücke
44 Aufgaben

45 **4 Variable und einfache Variablentypen**
45 Mehrfachverwendung einer Prozedur
48 Die Deklaration von Variablen
49 Die Wertzuweisung
50 Die einfachen Variablentypen
53 Etiketten mit Varianten
54 Der Datentyp STRING
56 Aufgaben

57 **5 Die FOR-Anweisung**
57 Ein erstes Beispiel zur FOR-Anweisung
59 Zählschleifen

61 Allgemeine Beschreibung
63 Syntaxdiagramme
65 Verbundanweisungen
66 Umkehrung der Durchlaufrichtung
67 Zerlegung von STRINGs
68 Aufgaben

70 **6 Dialogprogramme**
70 Das Palindrom-Beispiel
71 Funktionsweise von Read und ReadLn
72 Der Eingabedialog
73 Masken
75 Eine Maskenprozedur
76 Das Programmierschema E-V-A
78 Ausgabe auf dem Drucker
80 Aufgaben

82 **7 Rechnen mit dem Computer**
82 Addieren, Subtrahieren, Multiplizieren
84 Dividieren mit /
85 Die ganzzahlige Division mit DIV und MOD
86 Beispiel
88 Real und Integer
90 *Hin und zurück zwischen den Typen*
91 *Verbessertes Stückelungsprogramm*
92 Zahlen und Zeichen
94 *Von STRINGs zu Zahlen*
95 Ein neues Movie
97 Aufgaben

99 **8 Die Entscheidungsstruktur**
99 Sequenz und Entscheidungsstruktur
101 Die IF-THEN-ELSE-Anweisung
103 Bedingungsausdrücke
105 Syntax der IF-Anweisung
107 Programmablaufpläne und Struktogramme
109 *Nassi-Shneiderman-Diagramme*
110 Einseitige Entscheidungen und geschachtelte Strukturen
111 *Umgang mit geschachtelten Bedingungen*
113 Entscheidungs- und Wiederholungsbedingungen
114 Aufgaben

116 **9 Wiederholungsstruktur und boolesche Variable**
117 Der verpönte Sprung zurück
118 Die WHILE-Anweisung
121 *Ein Beispiel zur WHILE-Anweisung*
123 Initial- und Finalzustände einer Schleife
125 Boolesche Kontrollvariable für die Schleife
127 Die Verarbeitung des Zufalls
129 Ein kleines Würfelspiel
131 Simulation eines Ökosystems
137 Aufgaben

139 **10 Textdateien und STRING-Verarbeitung**
140 Dateien in Pascal
141 *Die Brücke zwischen Pascal-Programm und Betriebssystem*
142 *Dateien öffnen und schließen*
143 Abarbeiten einer Textdatei
145 Daten speichern und wieder holen
147 *Geräte als Dateien*
148 Text Generators Ltd.
150 *Wieder Zeilen erzeugen*
152 STRING-Spezialitäten
156 Das Silbentrennprogramm
161 *Zur Erklärung*
161 Aufgaben

163 **11 Menü-Steuerung von Programmen und Mehrfachauswahl**
163 Zur Gestaltung einer Menü-Oberfläche
164 Mehrfachauswahl mit IF-THEN-ELSE
167 Die Menü-Schleife
168 *Menüs mit Auswahl über Char-Werte*
170 Die CASE-Anweisung
172 Syntax der CASE-Anweisung
173 *Die „anderen" Werte bei einer CASE-Anweisung*
174 Einfache Eingabekontrollen und die REPEAT-Schleife
178 *Eine komfortablere Eingabekontrolle*
179 Die CASE-Anweisung außerhalb von Menüs
182 Aufgaben

183 **12 Aufzählungs-, Teilbereichs- und SET-Typen**
183 Werte, Variable und Typen
184 *Typ, Variable, Werte*
186 *Die Typdeklarationen im Gesamtaufbau eines Pascal-Programms*
188 Zur allgemeinen Syntax der Typdeklaration
189 Selbstdefinierte Aufzählungstypen
190 *Anwendung von selbstdefinierten Aufzählungstypen*

192 *Ord und die Funktionen bei Aufzählungstypen*
192 Teilbereichstypen
194 *Explizite und implizite Typdefinitionen*
195 SET-Typen
197 *SETs und ihre Elemente*
198 *Etwas SET-Theorie*
199 *Beispiel: Karten ziehen*
204 Aufgaben

205 **13 Prozeduren und Funktionen mit Parametern**
205 Die Einführung von Parametern
208 Die Unterscheidung lokaler und globaler Größen
210 Werte- und Variablenparameter
212 *Zwei, drei, viele Parameter*
213 Modularisierung
216 Prozeduren für den täglichen Gebrauch
218 Vier weitere nützliche Bausteine
222 UNITs
224 Funktionen
226 *Selbstdefinierte Funktionen*
228 Zur geschachtelten Verwendung von Prozeduren
229 Rekursion
231 Aufgaben

233 **14 RECORD, ARRAY und FILE**
233 Die Abbildung realer Objekte im Datenverbund
236 *Ansprechen der RECORD-Komponenten*
237 Die WITH-Anweisung
239 Speichern in Textdateien
240 *Das Beispiel für den Bücherfreund*
242 ARRAYs
244 *Computerisierung für das BuecherRegal*
246 Geschachtelte Datentypen
249 *Tief geschachtelte Datentypen*
251 Variante RECORDs
253 Die Datenstruktur FILE OF für den Direktzugriff auf Datensätze
255 Sortieren
258 Weitere Beispiele
260 *Programmieren eines Font-Editors*
262 Aufgaben

266 **Anhang: Der IBM-Zeichensatz**

268 **Register**

Einleitung

Pascal ist eine allgemeine Computersprache, mit der sich Aufgaben aus den unterschiedlichsten Bereichen auf Mikrocomputern wie auf Großrechnern bearbeiten lassen. Ihr Entwickler, der Schweizer Informatikprofessor Niklaus Wirth, hat besonderen Wert darauf gelegt, daß in ihr die grundlegenden Prinzipien des Programmierens deutlich zum Ausdruck kommen. Je nach Verwendungszweck haben viele Programmiersprachen ihre eigene Bedeutung; als Lernsprache ist Pascal unübertroffen. Allen, die sich tiefergehend praktisch und theoretisch mit der Programmierung von Computern befassen wollen, kann der Zugang über Pascal nur empfohlen werden.

Wir gehen von der Annahme aus, daß Sie einen der üblichen, heute sehr preiswert erhältlichen Mikrocomputer mindestens gelegentlich zum Üben nutzen können. Weiterhin sollte ein handelsüblicher Pascal-Compiler (das ist das Programm, das einen in Pascal geschriebenen Code in den Maschinencode übersetzt), bereits auf dem Rechner installiert sein bzw. wie im jeweiligen Handbuch beschrieben, installiert werden, damit Sie die Übungsbeispiele gleich ausprobieren können. Wegen der großen Verbreitung dieses Pascal-Dialekts werden Sie in diesem Buch die Pascal-Variante Turbo-Pascal für den IBM-PC und Kompatible kennenlernen, die von der amerikanischen Firma Borland entwickelt wurde. Spezielle Eigenschaften von Turbo-Pascal werden aber entweder gemieden oder nur so weit verwendet, wie sie sich mit geringfügigen Abwandlungen auch auf andere Pascal-Dialekte (zum Beispiel Quick-Pascal von Microsoft) übertragen lassen. Sie finden durchgängig die erforderlichen Hinweise auf die Unterschiede zwischen den – noch gelegentlich verwendeten – älteren Versionen von Turbo-Pascal (bis zu Version 3.x) und der Neugestaltung der Sprache (in den Versionen ab 4.0 und 5.5).

Das vorliegende Buch dient als qualifizierte didaktische Schulungsunterlage für den Einstieg ins Programmieren – nicht als Ersatzhandbuch mit unzusammenhängenden Befehlserläuterungen. Bedienungshinweise zu Grundfunktionen des Computers finden Sie in den Bedienungsanleitungen oder in einem der zahlreichen Grundlagenbücher. Bitte schlagen Sie dort nach, wenn Sie

beispielsweise wissen wollen, wie ein Diskettenlaufwerk funktioniert, wie man Dateien kopiert, Disketten formatiert und so weiter.

Die Übungsaufgaben

Jedem Kapitel sind reichlich Aufgaben beigefügt, an denen Sie Ihr Verständnis des Stoffs erproben und vertiefen können. Wenn Sie sich zirka 50 Prozent der vorgeschlagenen Aufgaben nach Ihren eigenen Interessen auswählen und sie sorgfältig bearbeiten, reicht dies aus, um die erforderliche Übung zu erlangen. Sicherlich werden Sie die eine oder andere Aufgabe nicht ganz verstehen oder nicht vollständig lösen können. Das ist ganz natürlich – denken Sie daran, daß Sie ein Programmierproblem nur dann erfolgreich angehen können, wenn Ihnen der Zusammenhang, aus dem es stammt, gut vertraut ist.

Ein ganz wichtiger Tip: Versuchen Sie auf keinen Fall, gleich zu Anfang ein großes Programm mit allem Komfort zu entwerfen, wie Sie ihn aus kommerziellen Programmen kennen. Sie würden es gewiß niemals zu Ende schreiben. Stattdessen sollten Sie zunächst immer anstreben, eine sorgfältig strukturierte, aber ansonsten möglichst primitive Lösung tatsächlich fertigzustellen. Verfeinerungen lassen sich bei guter Strukturierung später immer noch anbringen. Wundern Sie sich nicht über die Arbeit und Mühe, die das dennoch macht. Ein einziges Programm über einen komplexen Gegenstand erreicht rasch eine Länge von mehreren tausend Programmzeilen.

1 Grundlagen

Maschinensprache – Computercodes – Pascal – Der Editor und das Turbo-Hauptmenü

Maschinensprache

Das „Herz" jedes Computers ist sein Prozessor. Grob gesagt, erkennt der Prozessor bestimmte Bitmuster an seinem Ein-/Ausgangskanal und reagiert darauf. Ein Bit ist die kleinste Informationseinheit: Null oder Eins, Aus oder Ein, ... Die Bitmuster werden als Bytes oder „Computerwörter" bezeichnet, wobei der letztere Begriff recht unexakt ist.

Die Gesamtheit aller „Wörter" aus 0 und 1, die ein bestimmter Prozessor als Befehle interpretiert, nennt man den „Befehlssatz" des Prozessors. Aneinanderreihungen von Bytes, die solche Befehle und zugehörige Daten enthalten und den Computer zu einer Folge von genau definierten Aktionen veranlassen, bilden ein Programm in der Maschinensprache des betreffenden Computers. Bei einem ganz einfachen Computertyp könnte etwa folgendes „Wörterbuch" gelten:

Jede Vereinbarung über die Bedeutung von „Computerwörtern" wird Code (auch eingedeutscht Kode) oder Schlüssel genannt. Wenn die Computerwörter aus je drei Symbolen der zwei Zustände 0 oder 1 bestehen, wird von

Code	Befehl	Dezimalziffer
0 0 0	Pausieren	0
0 0 1	Daten holen	1
0 1 0	Addieren	2
0 1 1	Wiederholen	3
1 0 0	Subtrahieren	4
1 0 1	Vergleichen	5
1 1 0	Drucken	6
1 1 1	Bildschirmausgabe	7

Abbildung 1-1: Eine Wörterbuchtabelle

einem 3-Bit-Code gesprochen (ein Bit entspricht einer mit 0 oder 1 besetzbaren Position). Die meisten heutigen Computer arbeiten mit einem 8-Bit-Code mit maximal 256 verschiedenen „Wörtern" als Maschinensprache (wobei 8 Bit zu einem Byte zusammengefaßt werden). Signalfolgen aus je 8 Nullen und Einsen stellen also letztlich die „Sprache" dar, die der Computer versteht.

Computercodes

Der ASCII-Code und die PC-Zeichen

Neben dem Code für Befehle werden noch eine Reihe weiterer Kodierungen verwendet, die alle ebenfalls 8 Bit oder ein Vielfaches davon umfassen. Je nach Zusammenhang, je nachdem, welche „Datenweichen" vorher gestellt wurden, kann dieselbe Folge von Nullen und Einsen nach dem einen oder dem anderen Schema gedeutet werden. Die wichtigsten Kodierungen finden Sie in der Tabelle 1 im Anhang wiedergegeben. Sie bilden in ihrer Gesamtheit den erweiterten ASCII der IBM-kompatiblen Personal Computer. ASCII bedeutet „American Standard Code for Information Interchange".

Dual- und Hexadezimalsystem

Der andere Code, der sich in dieser Tabelle niederschlägt, ist die Umsetzung der im gewöhnlichen Dezimalsystem 0, 1, 2, 3, 4, 5, .. 255 geschriebenen Zahlen in die binäre Folge 0, 1, 10, 11, 100, 101, .. 11111111 (binär heißt „aus zwei Komponenten gebildet"). Da der Computer letztlich nur „Strom ein" und „Strom aus" kennt (so stellen sich Nullen und Einsen für ihn dar), werden intern Zahlen in dieser Form, das heißt im Zweier- oder Dualsystem dargestellt, das mit den beiden Symbolen 0 und 1 auskommt. Rechnen läßt sich mit den dualen Zahlen wie mit den dezimalen. Entscheidend ist, daß sich dieses Rechnen leicht mit Transistorschaltern imitieren läßt: denn aus diesen bestehen Mikroprozessoren hauptsächlich.

Schließlich sehen Sie in der Tabelle noch eine Spalte, die zu den Dezimal- beziehungsweise den Dualzahlen die Entsprechung im Sechzehner- oder sogenannten Hexadezimalsystem angibt. Die Bedeutung dieser Zahlendarstellung liegt darin, daß sie leichter zu schreiben und auch zu behalten ist als die duale Form, aber nach einem ganz einfachen Schema in die Zweierform übersetzt werden kann (was in der Regel der Computer erledigt).

„In den Bits wühlen" müssen Sie aber nur in seltenen Ausnahmefällen – in diesem Buch werden Sie kein Beispiel hierfür entdecken. Erst recht werden Sie niemals in Maschinensprache dem Computer Anweisungen geben. Nur in der Urzeit der Computerei mußten die Programmiererinnen und Programmierer die Geräte mit langen maschinensprachlichen Befehlsfolgen füttern – sofern die Schaltungen nicht gleich per Hand durch Umstecken von Leitungen auf einer großen Schalttafel eingerichtet wurden. Heute braucht man für die allermeisten Zwecke nicht so tief im Inneren der Computer hantieren. Zum Programmieren werden höhere oder problemorientierte Programmiersprachen (im Gegensatz zu den maschinenorientierten) verwendet – zum Beispiel Pascal.

Pascal

Die Entwicklung der höheren Programmiersprachen begann in den fünfziger Jahren. Die Programme sollten sich nicht länger mit den ganz speziellen Schaltvorgängen beim jeweiligen Computertyp befassen müssen. Sie sollten maschinenunabhängig werden, um sich desto besser der zu lösenden Aufgabe, dem Problem, zuwenden zu können. Die Zeit, die man damit verbringt, sich mit den Spezialitäten der Maschine zu plagen, ist für die Bearbeitung der eigentlichen Aufgabe vergeudet. Auch benutzt man beim Lösen von Problemen eine anders strukturierte Logik als jene, die den Computern zur Verfügung steht. Der Mensch kann sich nur mühsam in die Sprachen der Maschinenwelt hineindenken. Wie wäre es, wenn sich stattdessen der Computer ein wenig der menschlichen Vorstellungswelt nähern würde?

Genau dafür wurden die problemorientierten Sprachen geschaffen. Für den Menschen sind Programmtexte in diesen Sprachen leichter lernbar und klarer strukturiert als die direkten Maschinenbefehle. Andererseits müssen die höheren Sprachen so strikt und eng auf mögliche Aktionen des Computers festgelegt sein, daß der Computer sie sich selbst in seine eigene Maschinensprache übertragen kann.

Was ist ein Programm?

Ein Programm oder ein Programmcode ist erst einmal ein Text, der aus einer Folge von Anweisungen in einer Programmiersprache besteht. Dieser Text soll für den Menschen eine möglichst klare Bedeutung und Bezie-

hung zu einer Aufgabe haben. Auf der anderen Seite muß ihn die Maschine rein formal mit ihren beschränkten Handlungsmöglichkeiten interpretieren können.

Die Kunst des Programmierens besteht darin, in der Programmerstellung beiden Ebenen gerecht zu werden: den Inhalt angemessen und klar zu beschreiben und dabei formal einwandfrei zu arbeiten. Pascal, das 1969 extra zu Lehrzwecken durch Niklaus Wirth, Professor an der ETH Zürich, entwickelt wurde, eignet sich dazu ohne Zweifel. Benannt ist Pascal nach dem Konstrukteur einer der ersten Rechenmaschinen, dem Mathematiker und Philosophen Blaise Pascal (1623-1662). Ihre schweizerische Heimat kann die Sprache nicht verleugnen: Sie gilt als Musterbild an Überschaubarkeit, darüber hinaus als exzellent in ihrer Geradlinigkeit bei der Übersetzung in den Maschinencode. Aus unterschiedlichen Gründen hat sie sich zwar in der Industrie weniger durchgesetzt als andere Programmiersprachen, dient aber als Vorbild bei allen Neuentwicklungen problemorientierter Sprachen. Als die wahrscheinlich beste vorhandene Lehrsprache hat sie internationale Verbreitung in der Informatikunterweisung an Schulen und Hochschulen gefunden. Ihre Einsatzmöglichkeiten sind so vielfältig wie die Welt des Programmierens. Neuere Pascal-Dialekte unterstützen auch modernste Programmiermethoden wie zum Beispiel die „objektorientierte" (abgekürzt OOP).

Bevor Sie nun damit beginnen, Pascal-Programme zu schreiben – das heißt Texte, die der Computer mittels eines Übersetzungsprogramms (des Compilers) in Maschinensprache umwandeln kann – sollten Sie sich jedoch mit Ihrem Editor vertraut machen.

Die Programmierumgebung

Das Hilfsmittel, das erlaubt, einen Programmtext in den Computer zu tippen und ihn zu korrigieren, wird allgemein Editor oder Editorprogramm genannt. Ein Editor ist praktisch ein (relativ einfaches) Textverarbeitungsprogramm.

Generell können Sie sich in einen Editor oder ein Textverarbeitungsprogramm Ihrer Wahl einarbeiten, wenn es diesem Programm möglich ist, einen Text als ASCII-Datei abzuspeichern. Der jeweilige Pascal-Compiler liest und übersetzt oder kurz: „compiliert" dann die ASCII-Datei,

wenn sie einwandfrei programmiert ist. Moderne Pascal-Compiler verfügen über einen in eine Bedienungsoberfläche integrierten Editor beziehungsweise über eine integrierte Entwicklungsumgebung (neben der gewöhnlichen Kommandozeilenversion des Compilers). Häufig gibt es auch einen sogenannten „Debugger" (ein Dienstprogramm zur Fehlersuche in den Programmcodes) und andere Hilfsprogramme innerhalb der Entwicklungsumgebung des Compilers. Da das Turbo-Pascal-System in der PC-Welt einen verbreiteten Standard darstellt, wird im folgenden kurz auf die Bedienung von dessen Entwicklungsumgebung eingegangen.

Starten und Laden von Turbo-Pascal

Wenn die Diskette mit dem Turbo-System in Laufwerk A: steckt, können Sie das System aufrufen, indem Sie *Turbo* hinter die Bereitmeldung A: schreiben (und die Taste Return beziehungsweise Enter drücken). Ab der Version 4.0 ist es allerdings sinnvoll, das Turbo-System vorher auf einer Festplatte zu installieren. Folgen Sie dabei den Installationshinweisen und dem jeweiligen Handbuch. Außer der Bereitmeldung (C, C:\TP, C:\TP4 oder ähnliches) ändert sich beim Festplattenbetrieb am Aufruf des Turbo-Systems nichts (also erst in das angelegte Turbo-Verzeichnis wechseln, in dem sich die Startdateien nach einer Installation befinden, dann *Turbo* eingeben und Return drücken). Wenn alles gut geht, erblicken Sie bei Versionen bis zu 3.x auf dem Schirm das Signet des Herstellers (Abbildung 1-2).

Gefragt, ob Sie Fehlermeldungen hinzuziehen möchten (*Include error messages*), drücken Sie *Y* für Yes, Ja, (oder auch *N*, für No, das ist im Moment egal). Sie landen einen

```
TURBO Pascal system          Version 3.01A
                                     PC-DOS

Copyright (C)1983,84,85      BORLAND Inc.

Monochrome display

Include error messages (Y/N)?
```

Abbildung 1-2: Das Eröffnungsbild von Turbo-Pascal 3.01

Moment später im Zentrum des Turbo-Systems, dem Hauptmenü. Bei den 3er Versionen von Turbo-Pascal sieht das ungefähr so aus:

```
Logged drive: A
Active directory: \TURBO

Work file:
Main file:

Edit        Compile  Run   Save

Dir         Quit  compiler Options

Text:       0 bytes
Free: 63485 bytes
```

Abbildung 1-3: Das Hauptmenü von Turbo-Pascal 3.x

Ab der Version 4.0 erscheint nach dem Aufruf eine Bildschirmmaske mit zwei Textfenstern und der Referenzzeile sowie eine Zeile mit diversen englischen Begriffen (die sogenannte Menü-Leiste). Die integrierte Entwicklungsumgebung von Turbo-Pascal 5.0 meldet sich beispielsweise mit folgendem Eröffnungsbild.

```
File   Edit   Run   Compile   Options   Debug   Break/watch
                              Edit
 Line 1   Col 1   Insert   Indent     Unindent   C:NONAME.PAS

                 Turbo Pascal

                 Version 5.0
            Copyright (c) 1983, 1988 by
            Borland International, Inc.

                              Watch

 F1-Help  F5-Zoom  F6-Switch  F7-Trace  F8-Step  F9-Make  F10-Menu
```

Abbildung 1-4: Das Eröffnungsbild von Turbo-Pascal 5.0

Das Turbo-Hauptmenü

Das englische Wort menu heißt eigentlich „Speisekarte" oder „Speisetafel" (im Sinne einer Kreidetafel, auf der die Speisen offeriert werden). In der Datenverarbeitung wird das Wort im Sinne einer Liste von Wahlmöglichkeiten gebraucht. Auf dem Bildschirm werden dabei verschiedene Auswahlpunkte – die Kommandos beziehungsweise Arbeitsschritte – angezeigt, die Sie aktivieren können. Die Auswahl aus dem Angebot erfolgt in der Regel durch Tippen des besonders gekennzeichneten Buchstabens des entsprechenden Menü-Punktes (meist der erste).

Erklärt wird im folgenden nur die Erzeugung einer Datei für den späteren Programmcode.

Dateieröffnung bei den 3er Versionen

- Mit *L* für Logged drive (zu deutsch etwa: „das eingestellte Diskettenlaufwerk") können Sie angeben, auf welchem Laufwerk Texte gesucht und geschrieben werden sollen. Tippen Sie *l* und geben Sie das Laufwerk an, in das Sie Ihre Arbeitsdiskette eingelegt haben (wenn das noch nicht geschehen ist, holen Sie es nach).
- Mit *D* für Directory, „Inhaltsverzeichnis", können Sie sich (nach Bestätigen mit Return) den Inhalt der Diskette im gewählten Laufwerk anzeigen lassen.
- Geben Sie dann *W* ein. Sie dürfen sich anschließend wünschen, welchen Text sie erstellen beziehungsweise bearbeiten möchten (work – „Arbeit", file – „Datei"). Geben Sie einen Dateinamen ein wie:

```
BEISP1.TXT
```

Wenn es den Text beziehungsweise die Datei noch nicht auf Diskette gibt, erscheint die Meldung *New File*, „neue Datei".

- Jetzt können Sie gleich edieren, also endlich Programmtexte schreiben. Dazu ist nur noch *E* zu drücken – mit einem Mal ändert sich die Bildschirmausgabe, und es erscheint eine Kopfzeile über einem noch leeren „Blatt".

Das Hauptmenü ab der Version 4.0

Die aktuelleren Versionen des Turbo-Systems weisen eine sorgfältiger ausgearbeitete und übersichtlichere, oft

```
Line 1        Col 1        Insert        Indent        A:BEISP1.TXT
```

Abbildung 1-5: Die Menü-Leiste der 3er Versionen

tief gegliederte Menü-Struktur auf. Hier wie bei 3.x bereiten am Anfang vor allem die englischen Begriffe Schwierigkeiten. Nach dem Aufruf von Turbo-Pascal ab Version 4.0 gehen Sie zum Edieren folgendermaßen vor:

- Im Hauptmenü wählen Sie die Option *File* durch Bestätigen des hell unterlegten Wortes mit Return oder durch Drücken des Anfangsbuchstabens *f* (*F* oder auch *Alt* und *f*). Damit ziehen Sie sozusagen eine zugeordnete Spezialauswahltafel (ein sogenanntes Pull-down-Menü) herunter, die kurzfristig einen Teil des Bildschirms überdeckt.
- Unter der Option Change directory verbirgt sich die Möglichkeit, einen Zugriffspfad für die zu bearbeitende Textdatei anzugeben, speziell ein Laufwerk. Wenn Sie also das Laufwerk für den Programmtext wechseln wollen, tippen Sie C und geben Sie das Arbeitslaufwerk an.
- Die Option *D*irectory erlaubt Ihnen, das Inhaltsverzeichnis des eingestellten Pfades anzusehen.
- Wenn der Punkt *L*oad, „Laden", angewählt ist, geben Sie den Namen Ihrer Arbeitsdatei an.
- Die Escape-Taste *Esc* bringt Sie in allen Fällen wieder aus einem Untermenü heraus (sie fungiert als „Befreier" oder Abbruchtaste).
- Zurück im Hauptmenü können Sie dann mit der kleinen Pfeiltaste auf die Option *E*dit wandern (und mit Return bestätigen) oder einfach das *e* oder auch *Alt* und *e* drücken: Der Cursor blinkt dann im Textfenster, und Sie können dort einen Programmtext schreiben.

Die Ebenen der Computernutzung

Bevor es weitergeht, veranschaulichen Sie sich mit dem Diagramm auf Seite 19 noch einmal den bis jetzt zurückgelegten Weg.

Beim Starten des Computers führen in der Maschine automatisch ablaufende Routinen auf die Bedienungsebene des Betriebssystems. Von dort aus werden Programme, wie das Turbo-System, aufgerufen, die den Benutzer in deren eigene Welt entführen. Von der Ebene des Turbo-Hauptmenüs können Sie nach der Festlegung eines Namens für die Arbeitsdatei in den Editor gehen. Der Editor bildet dann wieder mit eigenen Regeln den Rahmen für die Nutzung des Computers zur Programmtexterfassung.

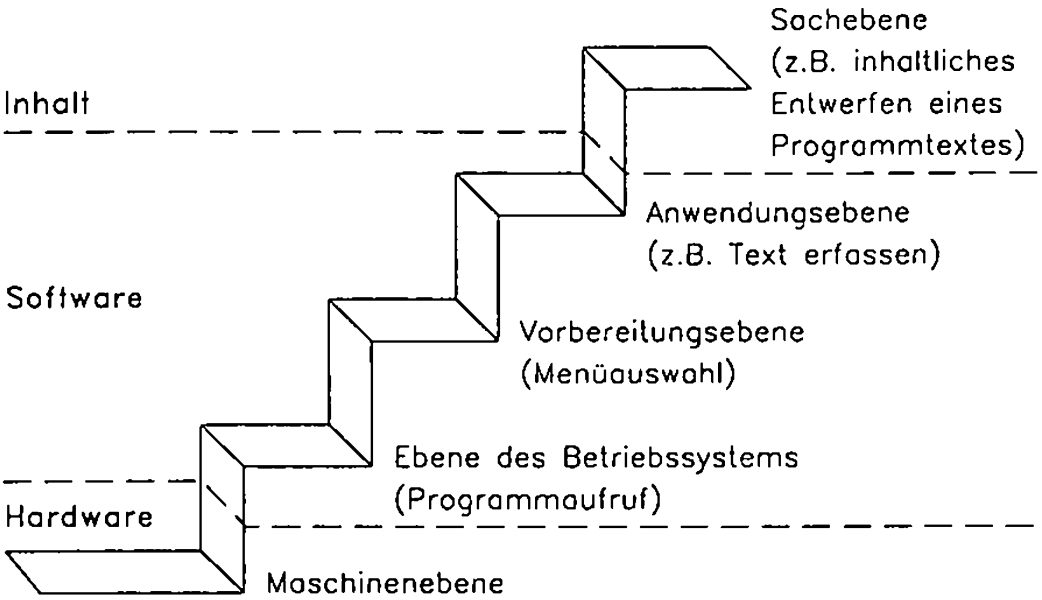

Abbildung 1-6: Stufenleiter der Computernutzung

Verlassen des Turbo-Editors

Aus dem Turbo-Editor kommen Sie mit der Tastenkombination *Ctrl-KD* heraus (Ctrl- bzw. Strg-Taste niederdrücken und halten; nun erst *k*, dann *d* anschlagen).

Nach dem Betätigen der Taste Return zeigt sich in Turbo 3.0, daß die Ebene des Turbo-Hauptmenüs wieder erreicht ist. Von dort zum Betriebssystem führt die Taste *Q* oder *q* für Quit, „Verlassen". Ab Turbo 4.0 können Sie die Entwicklungsumgebung über die Option Quit im File-Untermenü verlassen. Oder Sie können einfach die Tastenkombination *Alt-X* drücken, um zur Betriebssystemebene zurückzukehren.

Wordstar-Kompatibilität

Eine ausführliche Besprechung aller Befehle und Funktionen des Turbo-Editors würde an der Thematik des Buches vorbeigehen. Die Entwicklungsumgebung ab der Version 4.0 bietet zudem eine Hilfestellung (Taste *F1* – Help), mit der Sie jederzeit eine Erinnerungsstütze zu den Tastenbelegungen erhalten können. Nur soviel: Die meisten Befehle funktionieren nicht nur im Turbo-Editor, sondern ähneln den Tastenbelegungen einer großen Zahl von Textverarbeitungsprogrammen. Sie wurden zuerst im Textverarbeitungsprogramm Wordstar von MicroPro verwendet und vielfach nachgeahmt.

Die Texteingabe erfolgt im Editor wie bei einer Schreibmaschine. Am Ende jeder Zeile muß der „Wagenrück-

ein Zeichen nach links	Strg-S	←
ein Zeichen nach rechts	Strg-D	→
eine Zeile nach oben	Strg-E	↑
eine Zeile nach unten	Strg-X	↓
ein Wort nach links	Strg-A	
ein Wort nach rechts	Strg-F	
eine Bildschirmseite nach oben	Strg-R	Bild ⇑
eine Bildschirmseite nach unten	Strg-C	Bild ⇓
Springen zum Textanfang	Strg-QR	Strg Bild ⇑
Springen zum Textende	Strg-QC	Strg Bild ⇓
Zum Zeilenanfang	Strg-QS	Posl
Zum Zeilenende	Strg-QD	Ende
Springen zum Blockanfang	Strg-QB	
Springen zum Blockende	Strg-QK	
Zurück zur letzten Cursorposition	Strg-QP	
Zeichen links von Cursor löschen	Strg-H	⇐
Zeichen "unter" Cursor löschen	Strg-G	Entf
Wort rechts löschen	Strg-T	
Löschen der Zeile	Strg-Y	
Löschen bis zum Zeilenende	Strg-QY	
Block löschen	Strg-KY	
Blockanfang setzen	Strg-KB	
Blockende setzen	Strg-KK	
Block unsichtbar/sichtbar machen	Strg-KH	
Block kopieren	Strg-KC	
Block verschieben	Strg-KV	
Block auf Diskette schreiben	Strg-KW	
Datei als Block einlesen	Strg-KR	
Beenden Editieren	Strg-KD	
Umschalten Einfügen/Überschreiben	Strg-V'	Einfg
Suchen	Strg-QF	
Suchen und Ersetzen	Strg-QA	
Letzte Suche/Ersetzung fortführen	Strg-L	
Unterbrechung eines Befehls	Strg-U	Esc

Abbildung 1-7: Die Tastenbelegung des Turbo-Editors

lauf" (also *Return*) gedrückt werden, weil kein automatischer Zeilenumbruch vorgenommen wird. Die Tabelle (Abb. 1-7) zeigt eine Übersicht über die wichtigsten Funktionen und die Tastenbelegung des Turbo-Editors.

Mit der Bedienung des Editors, also des Werkzeugs, mit dem Sie später Programme aufschreiben, sollten Sie sich auf alle Fälle gründlich vertraut machen, bevor Sie

mit den ersten Programmbeispielen anfangen. Scheuen Sie nicht die mühsame Tipparbeit bei Übungstexten. An das Tippen müssen Sie sich ohnehin gewöhnen, da der Computer in der Regel seine Anweisungen über die Tastatur bekommt.

Aufgaben

1. Üben Sie Texte zu erfassen, zu korrigieren, zu speichern, sie wieder zu laden, sie zu verändern und wieder zu speichern – bis Ihnen der elementare Umgang mit dem Editor vertraut ist.
2. Finden Sie heraus, wie man kleine und große Textabschnitte („Blöcke") verschieben und kopieren kann, zum einen innerhalb eines Textes, zum anderen zwischen verschiedenen Dateien. Ihre Programme werden bald schon recht umfangreich werden. Dann können Sie sich die Schreibarbeit erheblich vermindern, wenn Sie weitgehend identische Abschnitte, eventuell sogar einzelne Wörter kopieren statt sie mehrfach einzutippen.
3. Für ganze Zahlen wird häufig ein 16-Bit-Code verwendet (zwei gekoppelte 8-Bit-Päckchen). Dabei wird das sechzehnte Bit (beziehungsweise das achte Bit des zweiten verwendeten Bytes) für das Vorzeichen reserviert. Versuchen Sie herauszufinden, bis zu welcher Zahl dieser Code reicht (die richtige Antwort finden Sie in einem späteren Kapitel).

2 Das erste Programm

Daten und Informationen – Ein neuer Ansatz – Das erste Pascal-Programm – Programmerklärung und Regeln – Aufgaben

Zeichen oder Zeichen von Zeichen benutzen wir nur, solange wir keinen Zugang zu den Dingen selbst haben.
Bruder William in: Eco, Der Name der Rose.

Das „Herz" des Computers besteht aus Transistor-Schaltern, die „0" und „1" verarbeiten. Doch wozu kann dies eingesetzt werden – was leistet der Computer im Prinzip? Die DIN-Norm 44300 sagt, daß ein Computer oder ein Datenverarbeitungssystem eine Funktionseinheit zur Verarbeitung von Daten, nämlich zur Durchführung mathematischer, umformender, übertragender und speichernder Operationen ist.

Von der Eingabe kommen Daten, die dann verarbeitet werden, so daß ein Resultat für die Ausgabe produziert wird. Zum Beispiel tippt man eine Rechenaufgabe in den Computer, und er errechnet das richtige Ergebnis. Oder die Polizei fragt beim Verkehrszentralregister an, wer der Halter eines falsch parkenden Kraftfahrzeugs ist, und bekommt von dem Computer eine Antwort. Da der Computer so viele verschiedenartige Aufgaben bewältigen kann, wird auch gesagt, daß der Computer ein universelles Gerät zur Datenverarbeitung oder auch zur Verarbeitung von Informationen ist.

Daten und Informationen

Was aber sind Informationen beziehungsweise Daten? In der alltäglichen und in der juristischen Sprache ist ihre Bedeutung klar: Als Information, als Daten werden alle Angaben, Kenntnisse oder Nachrichten über Personen,

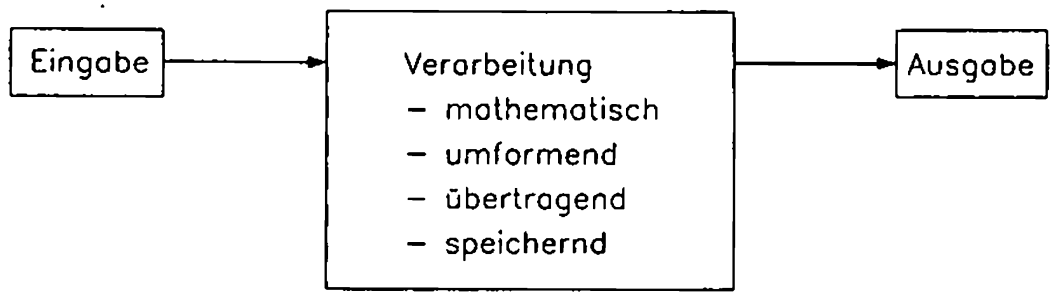

Abbildung 2-1: Das Computermodell nach der DIN-Norm

Sachen, Tatsachen oder Sachverhalte bezeichnet. Eine Information im vollen Sinn der alltäglichen Sprache gibt es nur für jemanden und über etwas. Und auch ein „Datum" (Einzahl von Daten) wird erst durch die Kenntnis seines sachlichen und persönlichen Bezugsrahmens (des Kontextes) aussagekräftig. Damit zum Beispiel „Note 5" zu einer aussagekräftigen Angabe wird, muß bekannt sein, wie das Notensystem aufgebaut ist. Indessen muß, wenn eine Note als Maß für eine Leistung angesehen werden soll, noch mehr berücksichtigt werden: Unter welchen Umständen die „Note 5" zustande kam und so weiter. Daten erhalten ihren Sinn erst aus ihrer Interpretation. Ein Diagramm verdeutlicht den Vorgang:

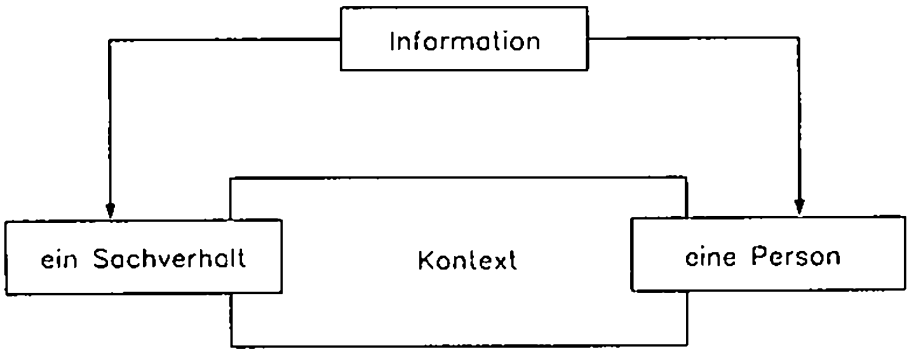

Abbildung 2-2: Eine Information teilt einer Person einen Sachverhalt aus einem bekannten Kontext mit.

Daten im Computer

Ein Computer kann aber im Gegensatz zum Menschen nicht be- oder verarbeiten, daß eine wirkliche Person etwas wissen soll und daß es ein konkretes Geschehen ist, über das Daten vorliegen. Bei der Dateneingabe in die Maschine wird der sachliche und persönliche Kontext abgeschnitten. Nicht mit den eigentlichen Informationen, sondern mit einem Skelett von ihnen wird der Computer gefüttert. Erst später kann ein Mensch, der die Daten und Informationen verwendet, wieder einen Bezug herstellen, das Skelett mit Leben erfüllen. Ein Mißverhältnis bei der Interpretation von Daten liegt heute oft darin, daß komplizierte Datenzusammenhänge von keiner einzelnen Person mehr wirklich überblickt werden.

Bei der Gestaltung von Computersystemen werden die Menschen, die die Anlage bedienen, unter den Begriff eines Anwendungsprozesses gefaßt, oder es wird von einer Schnittstelle zur Maschine gesprochen. Auch wenn die Menschen in solchen Modellvorstellungen wie spe-

zielle Computer erscheinen, darf nicht verkannt werden, daß Daten für den Menschen etwas völlig anderes sind als für den Computer.

Ein neuer Ansatz

Die Begriffsvielfalt und -entfremdung rührt zum Teil daher, daß in der mathematischen Informationstheorie ein rein formaler Informationsbegriff geprägt wurde, welcher die Informationsübermittlung mit der Übermittlung einer Folge von Zeichen gleichsetzt. Daß diese Vorstellung die Wirklichkeit des Computers trifft, kann man aus eigener Anschauung bestätigen.

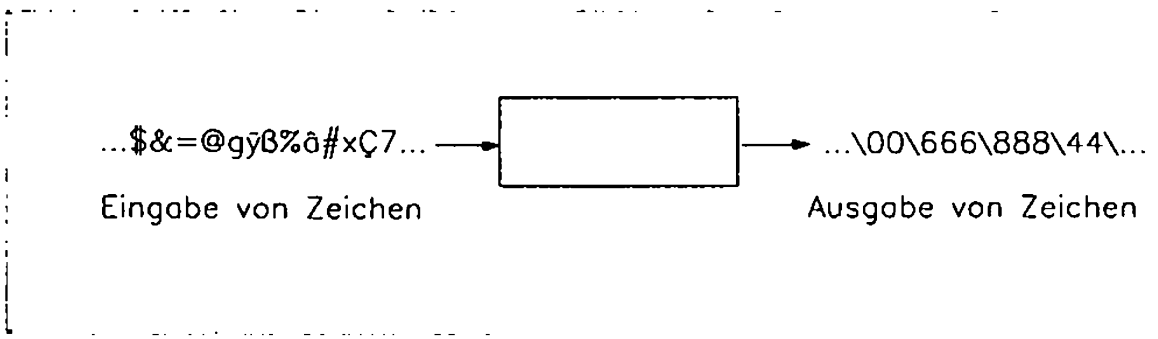

Abbildung 2-3: Computer schematisch

Die Tastatur als Eingabeschnittstelle bildet die Quelle eines Stroms von Buchstaben, von Ziffern und einigen anderen Zeichen. Der Bildschirm empfängt einen ebensolchen Strom und stellt ihn dar. Der Prozessor im Computer selber ist schließlich nur dasjenige Element, in dem die Verarbeitung (programmierte Auswertung und zielgerichtete Veränderung) der Zeichen stattfindet.

Ein Strom von Zeichen kann vielleicht einen ganzen Text oder eine komplexe Rechnung darstellen. Aber mit dem Sinn des Textes, ja selbst mit der zahlenmäßigen Be-

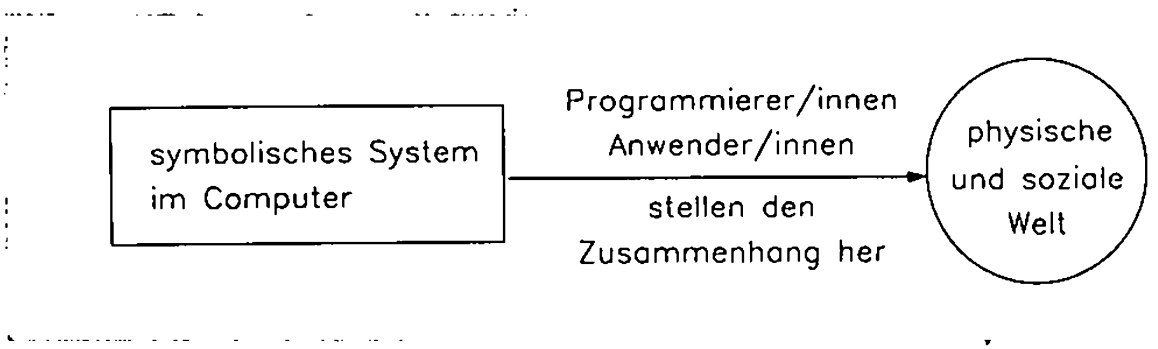

Abbildung 2-4: Welt und Computer

deutung der Ziffern, also dem ursprünglichen Informationsgehalt hat der Computer nichts zu tun. Der Computer be- und verarbeitet nur formale Zeichen. Hierin ist er allerdings äußerst flexibel und leistungsfähig. Man sagt, er ist eine universelle *Maschine zur Zeichenmanipulation.*

Der Programmierer oder die Programmiererin bestimmen dabei, nach welchen Regeln der Rechner aus eingegebenen Zeichen (oder Symbolen) andere Zeichen zu produzieren hat. Der Anwender oder die Anwenderin interpretiert, daß die Zahlen zum Beispiel Kontostände oder Rechtecke bedeuten sollen. Ein kleines Diagramm (Abbildung 2-4) verdeutlicht diese Zusammenhänge.

Programmieren

Eine universelle Maschine ist der Computer, weil er so ziemlich jedes andere Gerät, das Zeichen verarbeitet, nachahmen und simulieren kann. Dazu muß ihm eine passende Folge von Handlungsanweisungen aus einem vorhandenen Vorrat von möglichen Anweisungen gegeben werden. Die Gesamtheit der zulässigen Anweisungen hat einen Namen: es ist eine *Programmiersprache.*

In einer Programmiersprache kann eine Kette von Anweisungen gebildet werden. Eine zulässige Folge von Anweisungen an einen Computer in einer bestimmten Programmiersprache ist ein *Programm.*

Mit einem Programm richten Sie die universelle Maschine Computer so ein, daß sie als ein ganz spezielles Gerät arbeitet. Sozusagen bauen Sie sich eine Spezialmaschine, entsprechend Ihrem Wunsch – wobei Sie daran denken müssen, daß Ihre Wünsche sich nur im Bereich der Zeichen- oder Zahlenverarbeitung bewegen dürfen.

Das erste Pascal-Programm

Mit einem bescheidenen Wunsch beginnt das erste Pascal-Programm. Stellen Sie sich einen Druckereikasten vor, wie er als Kinderspielzeug verkauft wird. Dort stecken lauter einzelne Buchstaben, die zu Stempeln zusammengesetzt werden können. Ein Stempel, zum Beispiel ein Namensstempel, ist ein Gerät, das bei jedem Abdruck Zeichen von sich gibt, und zwar immer dieselben. Das kann man mit dem Computer nachbilden, der zum Beispiel Namen mit Hilfe des Bildschirms (oder des Drukkers) schreiben soll. Eine Schrift auf dem Bildschirm hat zwar nicht die Unverwechselbarkeit eines Stempelabdrucks mit seiner leicht verschmierten Farbe und anderen

individuellen Besonderheiten. Aber wenn man davon absieht und sagt, die wesentliche Eigenschaft des Stempels ist das Schreiben des Namens, dann kann man ihn im übertragenen Sinn durch den Computer ersetzen.

Abstraktion und Programmieren

Diese Überlegung ist typisch für die Anwendung des Computers. Man muß abstrahieren, um zwei an sich unterschiedliche Dinge als im wesentlichen gleich anzusehen. Vergleichen Sie eine Digitaluhr mit einer Ziffernblatt-Uhr. Beide zeigen die Zeit an. Aber ein rundes Ziffernblatt vermittelt ein ganz anderes Gefühl, was „fünf vor" oder „viertel nach" heißt, als die rein zahlenmäßige Angabe. Meist gibt es unterschiedliche Interessen, und je nach Interesse sind unterschiedliche Dinge wesentlich.

Zur Praxis

Im folgenden werden solche Überlegungen hintangestellt. Mit dem „Computer-Setzkasten" soll einfach etwas „gebastelt" werden, was man als Stempel deklarieren kann. Dazu müssen Sie dem Computer Anweisungen geben, daß er „wie ein Stempel" arbeiten soll. Das heißt, Sie müssen ein entsprechendes Programm schreiben. Ein solches Programm in der Programmiersprache Pascal sieht folgendermaßen aus:

```
PROGRAM Stempel;

BEGIN
 Write ('Rupert Röder');
 WriteLn;
 Write ('Pascalweg 123');
 WriteLn;
 WriteLn;
 Write ('65oo Mainz');
 WriteLn;
END.
```

Tippen Sie das Programm in den Turbo-Editor ein (mit *Ihrem* Namen und *Ihrer* Anschrift), speichern Sie es als STEMPEL.PAS-Datei ab und bringen Sie es (vom Hauptmenü aus) mit der C(ompile)-Option zur Compilation (das heißt zur automatischen Übersetzung in die Maschinensprache) und mit Hilfe von R(un) zum Ablauf. Dann erscheint auf dem Bildschirm:

```
Rupert Röder
Pascalweg 123

65oo Mainz
```

Programmerklärung und Regeln

Zur Erläuterung des ersten Programms sei zunächst angemerkt: Pascal kennt je nach Dialekt eine unterschiedliche Anzahl von Schlüsselwörtern oder auch Befehlen. Das Ihnen vorliegende Buch bezieht sich, wie bereits erwähnt, vorrangig auf Turbo-Pascal (weitgehend identisch mit Quick-Pascal). Eine Übertragung auf andere Pascal-Dialekte sollte indessen keine Schwierigkeiten bereiten. Obwohl die Schlüsselwörter im Programmcode groß oder klein geschrieben werden können, werden sie zur besseren Unterscheidung in diesem Buch generell groß geschrieben. Syntaxdefinitionen, Kommandozeilen-Parameter, Anweisungen und dergleichen (siehe unten) werden nur dann *kursiv* geschrieben, wenn es der Sinnzusammenhang erfordert. Folgende Schlüsselwörter sind in Turbo-Pascal reserviert:

Die Schlüsselwörter von Turbo-Pascal 6.0

ABSOLUTE, AND, ARRAY, ASSEMBLER, ASM, BEGIN, CASE, CONST, CONSTRUCTOR, DESTRUCTOR, DIV, DO, DOWNTO, ELSE, END, EXTERNAL, FAR, FILE, FOR, FORWARD, FUNCTION, GOTO, IF, IMPLEMENTATION, IN, INLINE, INTERFACE, INTERRUPT, LABEL, MOD, NEAR, NIL, NOT, OBJECT, OF, OR, PACKED, PRIVATE, PROCEDURE, PROGRAM, RECORD, REPEAT, SET, SHL, SHR, STRING, THEN, TO, TYPE, UNIT, UNTIL, USES, VAR, VIRTUAL, WHILE, WITH, XOR.

Programmname

Jedes Pascal-Programm fängt mit dem (englisch geschriebenen!) Schlüsselwort PROGRAM an. Danach kommt ein Name (im ersten Programmbeispiel *Stempel*), gefolgt von einem Semikolon. Der Name soll eine möglichst klare und treffende Überschrift für das Programm bilden. Die Namensvergabe will gut überlegt sein, denn treffende Bezeichnungen sind eines der wichtigsten Mittel, um sich und anderen beim Programmieren die Bedeutung der Vorgänge im Computer vor Augen zu halten und den Überblick nicht zu verlieren.

In Standard-Pascal wird hinter den Programmnamen (vor das Semikolon) noch *(Input, Output)* einschließlich der runden Klammern geschrieben. Es wird dadurch angedeutet, daß das Programm Eingaben und Ausgaben vollziehen soll.

Pascal-Wörter

Es gibt auch formale Vorschriften für die Erfindung von Namen oder Pascal-Wörtern. Bei der Bildung von

Namen (Pascal-Wörtern, identifier, auch „Bezeichner" genannt) gelten die folgenden Regeln:

- Ein Name muß mit einem Buchstaben beginnen.
- Danach dürfen Buchstaben und Ziffern in beliebiger Folge stehen.
- Auch der Unterstrich _ ist im Namen erlaubt, aber keine besonderen Zeichen wie der „Klammeraffe" @, die Tilde ~ oder ähnliches.
- Weil das ganze sich am amerikanischen Alphabet orientiert, zählen auch die deutschen Sonderzeichen (ä, ö, ü, Ä, Ö, Ü, ß) zu den nicht erlaubten Zeichen.
- Namen dürfen sehr lang sein (in Turbo-Pascal zum Beispiel maximal 127 Zeichen).
- Es dürfen Groß- und Kleinbuchstaben beliebig gemischt verwendet werden; zwei Namen, die nur in Groß- und Kleinschreibung voneinander abweichen, werden als identisch betrachtet.
- Ein vergebener Name kann in einem Programm nur in einer Bedeutung erscheinen.
- Die Schlüsselwörter von Pascal können nicht als Namen verwendet werden.

Richtig gebildete Namen sind zum Beispiel:

```
Nummer1
MeinAlterOnkel
Das5teBuchMose
Das_5_te_Buch_Mose
Beginn
```

Unzulässig sind die Namen:

```
007
Kaiser-Friedrich-Ring
Die8.teZahl
Änderung
Program
```

Erkennen Sie die Regelverstöße? (Bei *Program, program* oder, wie in diesem Buch geschrieben, PROGRAM handelt es sich beispielsweise um ein Schlüsselwort von Pascal, deshalb dürfen Sie es nicht als Namen verwenden.)

Anweisungen an den Computer

Die eigentlichen Anweisungen an den Computer werden eingeleitet durch das Schlüsselwort BEGIN und abgeschlossen durch END, wonach ein Punkt folgt. Die einzige Sorte von Anweisungen, die in dem kleinen Beispiel vorkommen, sind die *Write*- beziehungsweise *WriteLn*-Anweisungen.

Write bedeutet hier „Schreiben auf den Bildschirm". Was geschrieben werden soll, steht dahinter in runden Klammern. Wenn es sich um einen wörtlich zu übernehmenden Text handelt, so wird dieser nochmal zwischen Apostrophen gesetzt, das heißt zwischen zwei kleine Häkchen ' (auf PC-Tastaturen wird der Apostroph erzeugt, indem erst die Taste mit dem Auslassungszeichen gedrückt, dann die Leertaste betätigt wird).

WriteLn ist die Abkürzung für das englische Write line, „Schreibe eine Zeile". Ohne nähere Angaben bewirkt WriteLn einen Zeilenabschluß. Praktisch stellt es die Anweisung dar: „An den Anfang der nächsten Zeile vorrükken!". WriteLn kann auch mit beigefügtem Text in runden Klammern analog zu Write verwendet werden. Der Sprung in die nächste Zeile erfolgt, nachdem der Text auf den Bildschirm gebracht wurde. Nach den Anweisungen finden Sie stets einen Strichpunkt. Die Regel lautet:

Merkzettel

Zwei Anweisungen müssen durch Semikolon getrennt werden.

Weil es dieses Trennzeichen in Pascal gibt, haben Sie ansonsten – im Gegensatz zu anderen Programmiersprachen – beim Programmieren alle Freiheit. Sie können das Programm anordnen, wie Sie wollen, nur möglichst übersichtlich sollte es sein.

Grobgliederung eines Pascal-Programms

Ein Pascal-Programm besteht stets aus drei Teilen:

1. Dem Programmkopf
2. Dem Deklarations- oder Vereinbarungsteil
3. Dem Anweisungsteil

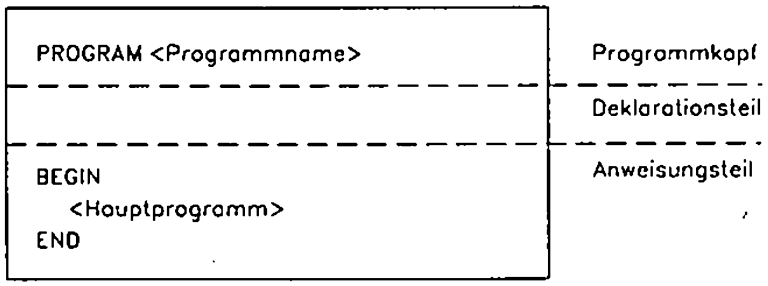

Abbildung 2-5: Grobgliederung eines Pascal-Programms

Der Programmkopf gibt dem Ganzen eine Überschrift. Im Anweisungsteil werden die Befehle an den Computer aufgerufen. Was macht der Deklarationsteil? Beim ersten Programmbeispiel enthielt er noch gar nichts. Jedoch wird er später bei großen Programmen weitaus den meisten Raum beanspruchen. Sie werden nach und nach sehen, was dort hinkommt. Vorab läßt sich schon sagen, daß dort Größen (Variablen) und Hilfsprozesse (Prozeduren) definiert werden, die nachher im Anweisungsteil (dem Hauptprogramm) verwendet werden können.

Aufgaben

1. Welche der folgenden Wörter sind zulässige Pascal--Namen?

```
Kantstraße_10
Zu-spaet-gekommen
DesMorgensInDerFruehe
NaechstesBeispiel
5terFall
Anfang
Programm
```

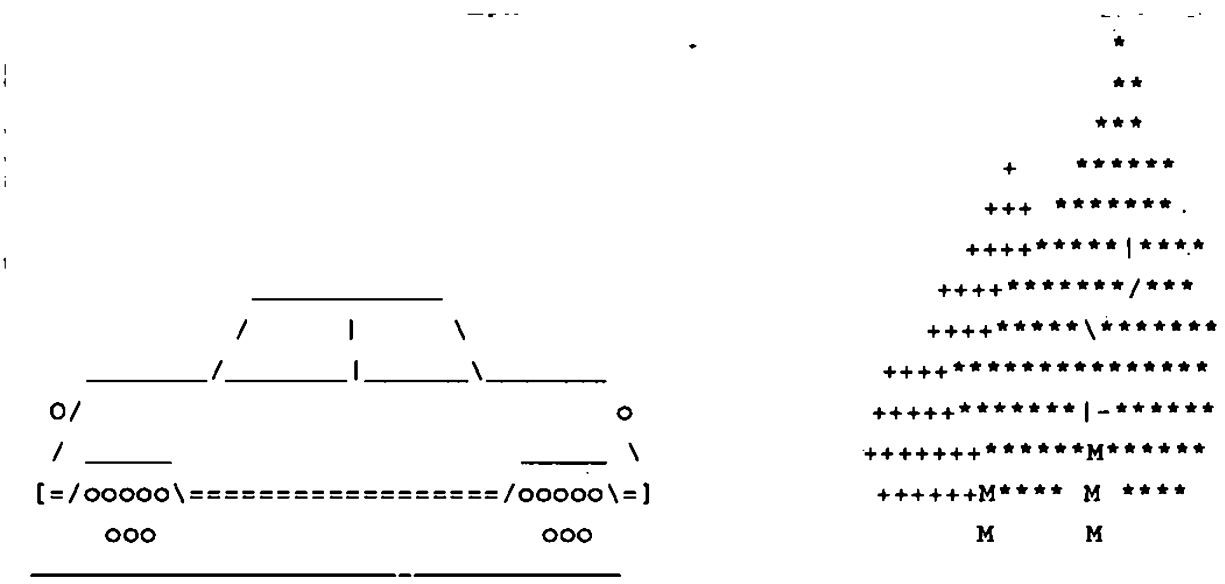

Abbildung 2-6: Auto und Baum in Zeichensatzgrafik

3 Prozeduren und Konstanten

Prozeduren – Lesen eines zusammengesetzten Pascal-Programms – Steueranweisungen für die Bildschirmausgabe – Feste Größen erhalten Namen – Konstanten für Zeichen und Textstücke

Teile und herrsche! (Altrömische Machtformel)

Das Programm des vorigen Kapitels hat auf dem Bildschirm quasi einen Stempel mit einer Adresse imitiert. Genauso können Sie ein Programm schreiben, das einen Rahmen auf den Bildschirm malt. Verwenden Sie hierzu die ASCII-Zeichen 179, 191, 192, 196, 217 und 218 . Sie erhalten diese, indem Sie die Ziffernfolge bei niedergedrückter Alt-Taste auf dem Zifferblock der Tastatur eingeben.

```
PROGRAM Rahmen;

BEGIN
  Write (' ┌──────────────────────┐');
  WriteLn;
  Write (' │                      │');
  WriteLn;
  Write (' │                      │');
  WriteLn;
  Write (' │                      │');
  WriteLn;
  Write (' │                      │');
  WriteLn;
  Write (' └──────────────────────┘');
  WriteLn;
END.
```

```
Rupert Röder
Pascalweg 123

65oo Mainz
```

Abbildung 3-1: Abdruck von GrosserStempel

Sie haben also ein Programm für den Druck der Adresse und eines für einen Rahmen. Wie lassen sie sich zu einem „Grossen Stempel" kombinieren, der die Adresse mit Umrahmung schreibt?

Prozeduren

Einer der Gründe, aus denen die Programmiersprache Pascal ihre große Bedeutung erlangt hat, ist die weitgehende Unterstützung, die sie für alle Verfahren bietet, ein größeres Programm aus Teil- oder Unterprogrammen aufzubauen. Zum Beispiel ist es möglich, aus den Programmen (kleiner) *Stempel* und *Rahmen* das Programm *GrosserStempel* zu erzeugen. Um ein ursprünglich selbständiges Programm zum Baustein eines größeren Programms zu machen, müssen Sie in Pascal folgendes tun:

- Das Schlüsselwort PROGRAM im Programmkopf durch das Schlüsselwort PROCEDURE ersetzen.
- Hinter das schließende END statt des Punktes ein Semikolon bringen.
- Die so entstandene Prozedurdefinition (oder Prozedurdeklaration) in den Deklarationsteil des übergeordneten Programms setzen.

Das übergeordnete Programm heißt dann Hauptprogramm, der Baustein Prozedur.

Aufruf einer Prozedur

Im Anweisungsteil des übergeordneten Programms kann die Prozedur mittels Nennung ihres Namens ein- oder mehrfach aufgerufen werden. Bei jedem Aufruf werden die in ihr enthaltenen Anweisungen der Reihe nach abgearbeitet.

Die einfachste Kombination der Unterprogramme *Stempel* und *Rahmen*, die nur leider noch nicht ganz das Gewünschte zeichnet, sieht so aus:

```
PROGRAM GrosserStempel;   {Erster Versuch}

PROCEDURE Stempel;
BEGIN
  Write ('Rupert Röder');

  WriteLn;
  Write ('Pascalweg 123');
  WriteLn;
  WriteLn;
```

```
  Write ('6500 Mainz');
  WriteLn;
END (* von Stempel *);

PROCEDURE Rahmen;
BEGIN
  Write ('                    ');
  WriteLn;
  Write ('                    ');
  WriteLn;
  Write ('                    ');
  WriteLn;
  Write ('                    ');
  WriteLn;
  Write ('                    ');
  WriteLn;
  Write ('                    ');
  WriteLn;
END (* von Rahmen *);

BEGIN (* von GrosserStempel *)
 Rahmen;
 Stempel;
END.
```

Kommentare in Pascal-Programmen

Sie finden hier zum ersten Mal Kommentare in das Programm eingefügt: zusätzliche Erklärungen, mit denen Sie für sich und andere das Programm klarer gestalten können. Damit der Computer sie beim Kompilieren überliest, müssen sie von geschweiften Klammern { } oder von (* am Anfang und *) am Schluß eingerahmt sein. In der Regel werden im folgenden die geschweiften Klammern verwendet, wenn es sich um eine rein inhaltliche Erläuterung zum Programm handelt. Die andere Klammernform ist nach formalen Gesichtspunkten gesetzt, zum Beispiel für die Angabe, welche Prozedur gerade endet.

Ein Kommentar, der mit der einen Klammerart eröffnet wurde, kann nur mit der dazugehörigen Schlußklammer aufhören! (Dadurch können sogar Kommentare der einen Sorte in Kommentaren der anderen stehen.)

Vergessen Sie, einen Kommentar zu schließen, oder haben Sie ihn mit der falschen Klammer nur vermeintlich verlassen, so erhalten Sie eine Fehlermeldung wie *unexpected end of source* („unerwartetes Ende der Quelldatei" beziehungsweise des Programms). Der Compiler findet

kein Programmende, weil Ihr vermutlich korrekt geschriebenes Schluß-END für ihn Teil des von ihm ignorierten, noch unabgeschlossenen Kommentars ist. Anders als in anderen Programmiersprachen können in Pascal auch ohne Kommentare schon viele inhaltliche Hinweise in das Programm eingeschlossen werden. Sparen Sie dennoch nicht mit zusätzlichen Angaben, wann immer es der Verständlichkeit dient.

Lesen eines zusammengesetzten Pascal-Programms

Was „macht" eigentlich das obige Programm? Um zu erfahren, was ein Pascal-Programm verrichtet, müssen Sie zunächst in den Anweisungsteil, genauer in den Anweisungsteil des Hauptprogramms, schauen. Denn auch die Prozeduren, die ja praktisch Programme sind, die ihre Selbständigkeit verloren haben, bestehen aus Prozedurkopf, Deklarations- und Anweisungsteil. Es muß also stets angegeben werden, von welchem Anweisungsteil die Rede ist. Damit die verschiedenen Teile leichter zu überblicken sind, kommentieren Sie am besten das abschließende END jeder Prozedur mit dem Prozedurnamen, eventuell ähnlich auch das BEGIN eines Anweisungsteils. Ein Pascal-Programm müssen Sie also stets „von hinten" lesen, da der Anweisungsteil am Schluß steht.

Wie Sie in dem Beispiel zum ersten Mal sehen, ist dieser in der Regel kürzer als der Deklarationsteil. Im Anweisungsteil von *GrosserStempel* wird als erstes die Prozedur *Rahmen* aufgerufen, die einen Kasten malt. Danach

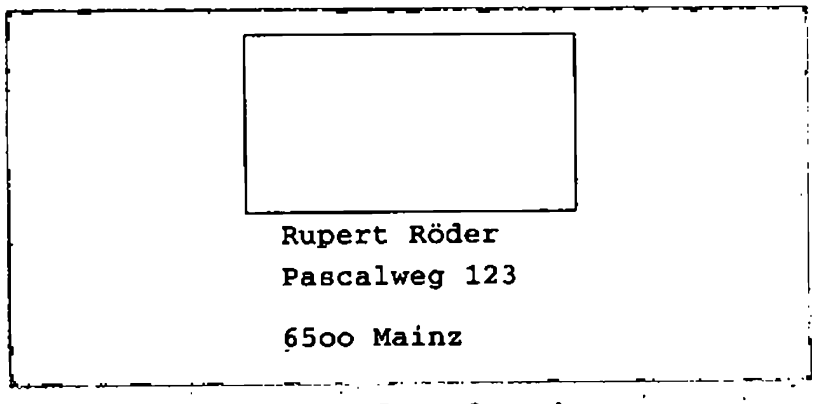

Abbildung 3-2: Abdruck von GrosserStempel

schreibt die Prozedur *Stempel* die Adresse auf den Schirm, und das Programm ist zu Ende.

Nur kann leider noch nicht die Rede davon sein, daß die Adresse in den Kasten gesetzt würde. Denn die WriteLn-Anweisungen, die beide Prozeduren verwenden, bringen ihre Texte alle brav untereinander. Das Programm erzeugt also nur ein Bild, das in etwa Abb. 3-2 (Seite 34) entspricht.

Steueranweisungen für die Bildschirmausgabe

Bildschirm „klar machen"

Als erstes ergänzen Sie das Programm durch die Anweisung: „Vor anderen Aktionen den Bildschirm leer machen". Diese Anweisung heißt in Turbo-Pascal *ClrScr*, was eine Abkürzung des englischen clear screen, „putze den Schirm", ist. Sie fügen die Anweisung hinter das BEGIN des Hauptprogramms ein. Nach dem Aufruf von ClrScr fängt ein Write oder WriteLn links oben in der Bildschirmecke an zu schreiben.

GoToXY

Damit sind Sie noch nicht viel weiter, denn das veränderte Programm setzt immer noch die Adresse unter statt in den Kasten. Sie müssen eine Möglichkeit finden, nachdem der Rahmen auf den Bildschirm gemalt ist, nochmal nach oben zu wandern und dort etwas hinzuzufügen. Diese Möglichkeit wird in Turbo-Pascal und einigen anderen Pascal-Varianten durch die GoToXY-Anweisung geboten (go to x y, für „gehe zu x y". Mit GoToXY kann die Schreibposition, die sonst zeilenweise nach rechts und unten vorrückt, an eine bestimmte Stelle des Bildschirms versetzt werden. Wohin gesprungen wird, ist durch zwei Zahlen, quasi einer x- und einer y-Koordinate anzugeben, also zum Beispiel:

```
GoToXY (20, 8)
```

Das GoToXY-Koordinatensystem

Dabei wird 20 als x- und 8 als y-Koordinate gewertet. Das Koordinatensystem sieht etwas anders aus als in der Mathematik: Es beginnt im Bildschirm in der Ecke links oben mit dem Punkt x = 1 und y = 1 (in Turbo-Pascal – hier gibt es manchmal Unterschiede zu anderen Pascal-Compilern). Wie sonst auch, beziehen sich die x-Koordinaten auf die waagrechte und die y-Koordinaten auf die senkrechte Achse.

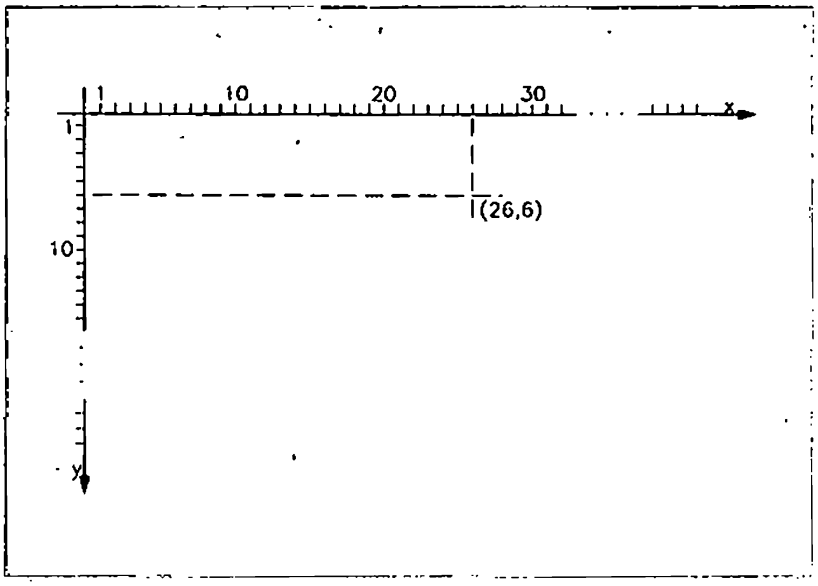

Abbildung 3-3: Das Koordinatensystem für GoToXY (in Turbo-Pascal)

Sprung zu Spalten und Zeilen

Einfacher ist es, sich folgendes zu merken: Die y-Koordinate bezeichnet die Nummer der Zeile auf dem Bildschirm, die x-Koordinate die Nummer der Spalte, und bei GoToXY wird in der Klammer erst die Spalte und dann die Zeile angegeben (durch Komma getrennt).

Gebrauch ab Turbo 4.0

Ab Turbo 4.0 müssen Sie in allen Programmen, die ClrScr oder GoToXY verwenden, angeben, daß Ihr Programm die Bildschirm-Hilfseinheit (englisch UNIT) mit speziellen Bildschirmbefehlen benutzt.

Sie müssen unmittelbar unter dem Programmkopf deklarieren:

```
USES crt;
```

beziehungweise

```
USES crt, printer;
```

(die UNIT *printer* erleichtert Ausgaben auf dem Drucker).

Damit Sie unter Kontrolle haben, was wohin auf den Schirm kommt, müssen Sie jetzt vor jedes Write eine GoToXY-Anweisung einfügen. Wenn Sie sich diese Arbeit machen, dann soll es gleich richtig sein, das heißt, Sie sollen noch ein weiteres Mittel kennenlernen, das die Sprache Pascal für das zuverlässige, komfortable und übersichtliche Programmieren anbietet: die Konstanten.

Feste Größen erhalten Namen

Erinnern Sie sich, was schon im vorigen Kapitel stand: Die sorgfältige Bezeichnung von Programmobjekten ist eines der wichtigsten Mittel, um sich und eventuell anderen die Bedeutung von Abläufen im Computer klar vor Augen zu halten. Sie müssen sich zum Grundsatz machen, nicht bloß das Korrekte zu meinen, sondern es auszusprechen beziehungsweise gewissenhaft niederzuschreiben. Ein Programm kann man nur auf Richtigkeit überprüfen, wenn seine Abläufe und deren Bedeutung nachvollzogen werden können. Dafür müssen die Bedeutungen möglichst deutlich und durchsichtig angegeben sein. Die Zahl 8 zum Beispiel kann für alles mögliche stehen: für einen Preis, einen Monat, eine Spalte, eine Zeile. Die jeweilige Bedeutung mit einem Namen zu versehen, ist häufig von Vorteil und zeugt von gutem Programmierstil. Sprechen Sie nicht von „Z" oder „Zahl", wenn Sie einen Monat meinen. Scheuen Sie auch nicht die Mühe, „Preis" statt „P" hinzuschreiben, wenn es sich um einen Preis handelt. Bei nächster Gelegenheit nehmen Sie „P" für Prozentsatz, verwechseln Preis und Preisänderung oder einen neuen mit einem alten Preis. Jede Ungenauigkeit des Ausdrucks trägt die Gefahr in sich, daß anschließende Überlegungen unklar werden.

Sie bestimmen nun in der endgültigen Fassung von *GrosserStempel*, was dort bei Ihnen die 8 sein soll. Nämlich die *Anfangszeile* der eingerahmten Adresse auf dem Schirm.

Konstantendeklaration

Wenn im Programm eine feste Zahl (oder auch andere feste Größen) vorkommt, die eine bestimmte Bedeutung trägt, definiert man sie als eine Konstante. Dafür schreiben Sie in den Deklarationsteil (nach Standard-Pascal-Vorschrift direkt hinter dem Programmkopf) unter der Überschrift CONST eine Konstantenvereinbarung, welche die Form

```
Pascal_Name = {Ein fester Wert};
```

hat. Zum Beispiel fügen Sie in das Programm *GrosserStempel* ein:

```
CONST
  Anfangszeile   =  8;
  Anfangsspalte = 20;
```

Besonders leistungsfähig werden die Konstanten dadurch, daß Sie andere Werte aus ihnen durch Addition (oder andere übliche Rechenoperationen) ableiten können. Sie bilden feste Bezugspunkte für ein ihnen zugeordnetes System. Das fertige Programm sollte insgesamt so aussehen:

```
PROGRAM GrosserStempel;
   {In perfekter Form}

USES crt, printer;      {Muß nur ab den Versionen
Turbo-Pascal 4.0 eingefügt werden.

CONST
  Anfangszeile   =  8;
  Anfangsspalte = 20;

PROCEDURE Stempel;
  BEGIN
    GoToXY (Anfangsspalte + 4, Anfangszeile + 1);
    Write  ('Rupert Röder');
    GoToXY (Anfangsspalte + 4, Anfangszeile + 2);
    Write  ('Pascalweg 123');
    GoToXY ( Anfangsspalte + 4, Anfangszeile + 4);
    Write  ('65oo Mainz');
  END (* von Stempel *);

PROCEDURE Rahmen;
  BEGIN
    GoToXY (Anfangsspalte, Anfangszeile + 0);
    Write ('┌──────────────────────┐');
    GoToXY (Anfangsspalte, Anfangszeile + 1);
    Write ('|                      |');
    GoToXY (Anfangsspalte, Anfangszeile + 2);
    Write ('|                      |');
    GoToXY (Anfangsspalte, Anfangszeile + 3);
    Write ('|                      |');
    GoToXY ( Anfangsspalte, Anfangszeile + 4);
    Write ('|                      |');
    GoToXY (Anfangsspalte, Anfangszeile + 5);
    Write ('└──────────────────────┘');
  END (* von Rahmen *);

BEGIN (* von GrosserStempel *)
  ClrScr;
  Rahmen;
  Stempel;
END.
```

Wenn Sie das Programm in den Computer tippen, können Sie reichlich Gebrauch von der Kopierfunktion des Editors machen, da viele Zeilen fast identisch sind. Übrigens: Bei den meisten Computern gibt es eine Taste, *PrtSc* (Print screen, „Drucke den Bildschirm aus") oder einfach *Druck* beschriftet, mit der Sie einen Abzug des momentanen Textbildschirminhalts auf dem Drucker erhalten, sofern einer angeschlossen ist. Wenn das Programm läuft, können Sie sich auf diese Weise sein Ergebnis ausdrucken lassen.

Änderbarkeit durch Konstanten

Finden Sie nicht auch, daß das Programm in dieser Form viel verständlicher ist, als wenn in der GoToXY-Anweisungen lauter Zahlen stehen, deren Bedeutung nicht direkt angegeben ist? Sie beobachten den zweiten Vorteil, der mit der reichlichen Verwendung von Konstanten verbunden ist: Wenn Sie eine neue Idee haben, wohin die Figur gezeichnet werden soll, zum Beispiel soll sie in die Ecke rechts oben kommen oder richtig in die Mitte gerückt werden, dann müssen Sie jetzt nur noch die Definition der Konstanten am Programmanfang abändern. Ohne Konstanten hätten Sie das ganze Programm zu durchforsten und überall die Parameter von GoToXY neu zu berechnen und anzugeben. Konstanten dienen der sorgfältigen Bezeichnung von Bedeutungen in Pascal-Programmen und zugleich einer bequemen Anpassung an neue Anforderungen.

Bottom-up-Verfahren

Als Sie von den vorhandenen Bausteinen *Stempel* und *Rahmen* ausgingen und daraus das Hauptprogramm zusammensetzten, haben Sie das Verfahren angewandt, das in der Theorie der Programmentwicklung Bottom-up-Methode heißt, die „vom Boden nach oben" gerichtet ist. Diese Methode kann grob in die Irre führen. Denn da die Teilprogramme nicht für das größere Ziel konzipiert wurden, wäre es kein Wunder, wenn sie nachher auch nicht korrekt arbeiteten. Im kleinen Maßstab erleben Sie dies vielleicht, wenn Sie tatsächlich versuchen, Programme wie *Rahmen* und *Stempel* zusammenzuführen, und die Adresse nicht in die Umrandung paßt. Die Theorie lehrt deshalb, daß beim Programmieren zunächst vom Oberziel auszugehen sei und dann die Bausteine dafür zu suchen sind.

Top-down-Verfahren

Das Gegenstück zum Bottom-up-Verfahren heißt Top-down-Methodik. Gemäß dieser haben Sie zuerst die Spitze (top) und das Hauptziel zu definieren. Die Analyse dieses Ziels zeigt, welche Etappen, welche Zwischenziele gesteckt werden müssen, um das Oberziel letztlich zu erreichen. Auf die Etappen wenden Sie das gleiche Verfahren an, bis am Ende der Analyse leicht ausführbare Einzelschritte stehen. Auf das Programmieren in Pascal bezogen, bedeutet dies:

Die Aufgabe eines größeren Programms wird solange auf Prozeduren verteilt, welche sich wiederum in Teilprozeduren gliedern, bis schließlich die Teilprozeduren der letzten Ebene mit elementaren Pascal-Anweisungen wie Write, WriteLn und anderen zu realisieren sind.

Um die Zusammenhänge darzustellen, existiert eine eigene Form von Diagrammen, Strukturdiagramm oder Problemaufbaugraph genannt.

Schematisch sieht ein solches Diagramm ungefähr so aus:

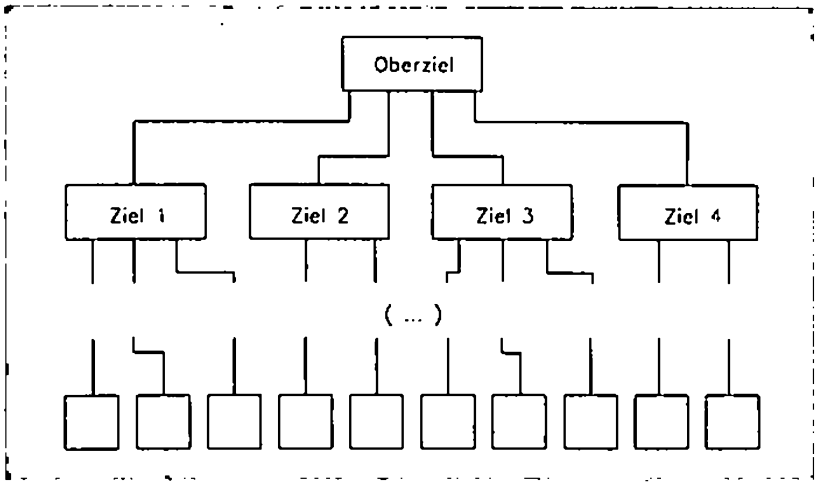

Abbildung 3-4: Problemaufbaugraph oder Strukturdiagramm

Schrittweise Verfeinerung

Es wird auch von dem Verfahren der schrittweisen Verfeinerung oder *stepwise refinement* gesprochen: Vom groben Anfangsziel ausgehend gelangen Sie mit der erforderlichen Zahl von Zwischenschritten schließlich zu Detailanweisungen. Völlig rein läßt sich das Verfahren aber nicht praktizieren. Schon dadurch, daß es auf einen bestimmten Vorrat von Detailhandlungen führen muß, trägt es ein Bottom-up-Element in sich. Wenn eine strenge Top-down-Methodik möglich wäre, müßte sich letztlich

jedes Ziel durch geeignete Zwischenschritte auf die bekannten Einzelhandlungen reduzieren lassen (eine allzu simplifizierende Annahme). In der Praxis herrscht zwangsläufig das Wechselspiel zwischen den großen Wünschen und dem Wissen, was im Detail machbar ist. Beides entwickelt sich in Abhängigkeit vom anderen fort.

Hierarchische Strukturierung von Programmen

In Entsprechung zu bürokratischen, militärischen oder anderen Organisationstrukturen sagt man, daß ein Programm entsprechend dem Top-down-Aufbau eine hierarchische Struktur aufweist. Das hat die ganz konkrete und für die Praxis wichtige Bedeutung, daß der Ablauf des Programms auf beliebig hoher Ebene verfolgt werden kann. So wie man in einer Organisation den Leiter einer Abteilung fragen kann, was in seinem Bereich geschieht, und er sich dafür verantworten muß, so genügt in einem hierarchisch strukturierten Programm die Betrachtung der Teilprogramme höherer Stufe, um einen Überblick über die Aktionsfolge des Programms zu erhalten.

Zur Praxis

Programmtechnisch entspricht die Zerlegung eines Ziels in seine Unterziele der Abfolge der Prozeduraufrufe im Anweisungsteil. Im Deklarationsteil werden dann die Prozeduren beschrieben, das heißt wiederum ihre Aufgaben auf Teilaufgaben zurückgeführt. Nicht ganz zu dem Schema der Top-down-Analyse paßt, daß Prozeduren auch mehrfach und in unterschiedlichen Zusammenhängen verwendet werden können.
Noch einmal zurück zum Beispiel *GrosserStempel*. Ein Strukturdiagramm dafür ist schnell gezeichnet:

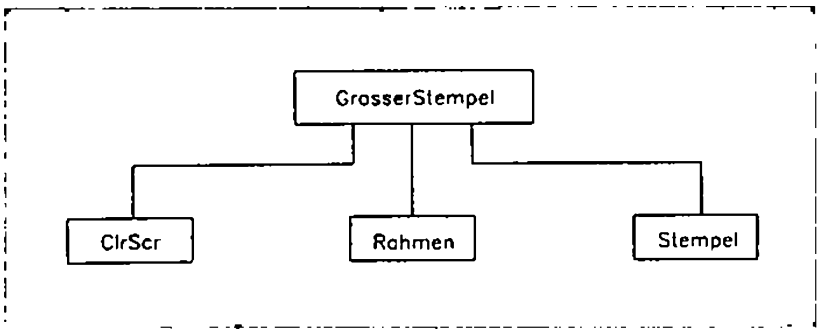

Abbildung 3-5: Problemaufbaugraph GrosserStempel

Genau genommen müßten die Teilaufgaben *Rahmen* und *Stempel* wieder untergliedert und auf ihre Einzelanweisungen (Write, ...) zurückgeführt werden. Zum Schluß des Kapitels soll mit einer kleinen Abwandlung des Stempelprogramms demonstriert werden, wie man Konstanten auch für andere Sorten von Größen als Zahlen anwendet.

Konstanten für Zeichen und Textstücke

Oben sahen Sie schon, wie die bloße Änderung der Konstantendefinition eine Korrektur an vielen Stellen des Programms ersparen kann. Dies läßt sich ausnutzen, um den *Rahmen* unterschiedlich gestalten zu können. Dazu fügen Sie im Programm unter die Definition der Konstanten *Anfangszeile* und *Anfangsspalte* noch eine Zeile ein, in der eine Konstante für das *Rahmenzeichen*, das heißt das Zeichen, aus dem der Rahmen bestehen soll, vereinbart wird. Ausnahmsweise bildet hier nur ein einzelner Buchstabe die Konstantenbezeichnung, weil sonst die Darstellung unübersichtlich würde. Probieren Sie es zunächst mit

```
CONST
  Anfangszeile  =   8;
  Anfangsspalte =  20;
  Z             = '*';     {RahmenZeichen}
```

Sie sehen: Wenn eine Konstante für ein einzelnes Zeichen, hier das Sternchen, oder wenn sie für eine Folge von Zeichen (zum Beispiel Pfeil = '—>') stehen soll, so muß das einzelne Zeichen oder die Zeichenfolge in Apostrophen gesetzt werden. Im Grunde handelt es sich um ein kurzes Textstück (ähnlich wie die Textstücke bei der Write-Anweisung).

Die Prozedur *Rahmen* wandeln Sie jetzt so ab, daß die vielen Zeichen, die den Rahmen bilden sollen, nicht direkt angegeben werden, sondern dafür eine Konstante verwendet wird. In die Write-Anweisung muß sie so oft aufgenommen werden, wie ein entsprechendes Zeichen stehen soll. Dabei lernen Sie gleich die Regel kennen:

Merkzettel

> Wenn mehrere Textstücke mit einer einzigen Write/WriteLn-Anweisung ausgegeben werden sollen, so wird alles in die Klammer geschrieben und durch Kommas voneinander getrennt.

Die Prozedur *Rahmen* sieht dann so aus:

```
PROCEDURE Rahmen;
  BEGIN
    GoToXY (Anfangsspalte, Anfangszeile + 0);
    Write (Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z);
    GoToXY (Anfangsspalte, Anfangszeile + 1);
    Write (Z,'                    ',Z);
    GoToXY (Anfangsspalte, Anfangszeile + 2);
    Write (Z,'                    ',Z);
    GoToXY (Anfangsspalte, Anfangszeile + 3);
    Write (Z,'                    ',Z);
    GoToXY (Anfangsspalte, Anfangszeile + 4);
    Write (Z,'                    ',Z);
    GoToXY (Anfangsspalte, Anfangszeile + 5);
    Write (Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z);
  END (* von Rahmen *);
```

Das Programm sollte folgendes Bild zeichnen:

```
************************
*   Rupert Röder       *
*   Pascalweg 123      *
*                      *
*   65oo Mainz         *
************************
```

Abbildung 3-6: Abdruck von GrosserStempel in Zeichensatzgrafik

Durch entsprechende Änderung der Konstanten können beliebige andere Zeichen an die Stelle des Sternchens treten. Statt das betreffende Zeichen in Apostroph hinzuschreiben erlaubt Turbo-Pascal auch, hinter einem Nummernzeichen # die Codenummer des Zeichens nach der ASCII-Tabelle anzugeben. Das ist vor allem für die Zeichen interessant, die zwar auf dem Bildschirm oder einem anderen Gerät ausgegeben werden können, aber im Editor nicht direkt zugänglich sind (im IBM-Zeichensatz zum Beispiel das Herz ♥ mit der Nummer 3). Mit Z = #3 beziehungsweise Z = ∞ erhalten Sie dann die folgenden Variationen des vom Programm erzeugten Bildes:

```
♥♥♥
```

4 Variable und einfache Variablentypen

Mehrfachverwendung einer Prozedur – Die Deklaration von Variablen – Die Wertzuweisung – Die einfachen Variablentypen – Etiketten mit Varianten – Der Datentyp STRING – Aufgaben

Im Wechsel steckt ein beständiger Geist.
F. Schiller

Nicht alles muß fest bleiben in einem Programm, sondern anstelle konstanter Werte können Platzhalter stehen, die während des Programmablaufs unterschiedliche Werte erhalten können. In der Informatik wird dann von Variablen gesprochen. Technisch gesehen entspricht einer Variablen ein (automatisch zugewiesener) kleinerer oder auch größerer Abschnitt im Speicher des Computers – sozusagen ein Fach, in dem bestimmte Dinge abgelegt und nachgesehen werden können. Allerdings gibt es in Pascal für verschiedene Arten von Daten, die im Speicher deponiert werden sollen, unterschiedliche Arten von Fächern, verschiedene *Typen* der Variablen. Der Typ einer Variablen ist sozusagen ihr „beständiger Geist", das, was sich bei der Variation der Werte nicht ändert. Durch die wechselnden möglichen Belegungen von Variablen kann ein- und dieselbe Folge von Anweisungen innerhalb eines Programms mehrere Aufgaben erledigen. Dieselbe Prozedur läuft immer wieder ab. Dennoch gibt es Abwandlungen bei dem, was geschieht.

Mehrfachverwendung einer Prozedur

Wenn Sie ein Programm mit Hilfe von Prozeduren *Waende, Dach, Tuer* und *Fenster* geschrieben haben, welches ein Bild ungefähr wie das folgende zeichnet, so möchten Sie vielleicht noch ein zweites Fenster an anderer Stelle hinzufügen.

Nichts einfacher als dies, sofern es im Programm Konstanten zur Bezeichnung von Anfangsspalte und -zeile des Fensters (beziehungsweise der relativen Anfangsspalte und Anfangszeile bezüglich eines festen Punktes des Hauses) gibt. Im folgenden Auszug aus einem Programmtext (den Sie an den Stellen, an denen Kommenta-

```
                                        O  O
                                       O O
                                      O
                                    IIIII
                                    I   I
                                    I   I
              DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
             DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
            DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
           DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
          DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
         DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
        DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
       DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
      DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
     DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
    DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
   DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
  DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
 DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DH                                                          HD
 H                                                           H
 H                                                           H
 H                                                           H
 H                                                           H
 H                                                           H
 H                                                           H
 H            HHHHHHHHHHH                  HHHHHHHHHHH       H
 H            H         H                  H         H       H
 H            H         H                  H         H       H
 H            H         H                  H         H       H
 H            H         H                  H         H       H
 H            H         H                  H         H       H
 H            H—        H                  H         H       H
 H            H         H                  MMMMMMMMMMMM      H
 H            H         H                                    H
 H            H         H                                    H
 H            H         H                                    H
 H            H         H                                    H
WWWWWWWWWWWWH————————-HWWWWWWWWWWW
```

```
PROGRAM Haus;

USES crt, printer;

CONST
  AusgangszeileHaus  = 20;{Untere linke Ecke}
  AusgangsspalteHaus = 15;{Untere linke Ecke}
                          {Weitere Konstanten ... }

VAR
  AnfangszeileFenster  : Integer;
                          {Unten links, relativ zum Haus}
  AnfangsspalteFenster : Integer;
                          {Unten links, relativ zum Haus}

PROCEDURE Waende;
  BEGIN
                          { ... }
  END (* von Waende *);

PROCEDURE Dach;
  BEGIN
                          { ... }

 END (* von Dach *);

PROCEDURE Tuer;
  BEGIN
                          { ... }
  END (* von Tuer *);

PROCEDURE Fenster;
  BEGIN
    GoToXY (AusgangsspalteHaus + AnfangsspalteFenster,
              AusgangszeileHaus - AnfangszeileFenster);
    Write  ('MMMMM');
    GoToXY (AusgangsspalteHaus + AnfangsspalteFenster,
              AusgangszeileHaus - AnfangszeileFenster - 1);
    Write  ('H   H');
    GoToXY (AusgangsspalteHaus + AnfangsspalteFenster,
              AusgangszeileHaus - AnfangszeileFenster - 2);
    Write  ('H   H');
    GoToXY (AusgangsspalteHaus + AnfangsspalteFenster,
              AusgangszeileHaus - AnfangszeileFenster - 3);
    Write  ('H   H');
    GoToXY (AusgangsspalteHaus + AnfangsspalteFenster,
              AusgangszeileHaus - AnfangszeileFenster - 4);
    Write  ('HHHHH');
  END (* von Fenster *);
```

```
BEGIN (* von Hauptprogramm *)
  ClrScr;
  Waende;
  Dach;
  Tuer;
  AnfangszeileFenster  :=  3;
                          {3 Zeilen über dem Boden des Hauses}
  AnfangsspalteFenster := 18;
  Fenster;
  AnfangsspalteFenster := 28;
  Fenster;
END.
```

Die Deklaration von Variablen

Am Beispiel ist zu sehen: Ähnlich wie Konstanten müssen auch Variablen deklariert werden. Die Variablendeklarationen oder -vereinbarungen stehen im Deklarationsteil des Programms unter der Überschrift VAR.

Variable und Konstanten in Prozeduren

Wenn ein Programm seine Selbständigkeit verliert und zur Prozedur eines übergeordneten Programms wird, darf es in Pascal seine eigenen Variablen und Konstanten behalten. Die Variable und Konstanten, die im Deklarationsteil einer Prozedur vereinbart sind, gelten als „privat". Sie sind nur innerhalb der Prozedur (einschließlich all ihrer Unterprozeduren) bekannt und können außerhalb nicht verwendet werden. Ihre Inhalte gehen verloren, wenn die Prozedur verlassen wird. Umgekehrt sind aber Variablen und Werte des Hauptprogramms (oder einer übergeordneten Prozedur) der Prozedur durchaus zugänglich. Daher können im Hauptprogramm zum Beispiel *AnfangszeileFenster* und *AnfangsspalteFenster* vereinbart sein und an die Prozedur *Fenster* überreicht werden.

In Standard-Pascal folgt die Vereinbarung von Variablen auf die Festlegung der Konstanten; es darf nur je einmal eine Überschrift CONST und eine Überschrift VAR geben. Turbo-Pascal erlaubt auch die Mischung der verschiedenen Abschnitte des Deklarationsteils, erkenntlich an der jeweils vorangestellten Überschrift.

Die Deklaration einer Variablen besteht aus der Nennung eines Namens, gefolgt von einem Doppelpunkt und der Angabe des Typs, den die Variable haben soll.

Typ

Der Typ kennzeichnet den Bereich der möglichen Werte, den die Variable annehmen kann. Wie Sie eine Konstante auf einen einzelnen Wert festlegen, so müssen Sie

bei Variablen entscheiden, für welche *Sorte* von Werten sie genommen werden soll. Wenn mehrere Variablen des gleichen Typs vorkommen, dürfen ihre Namen auch, durch Kommas getrennt, aufgezählt werden. Sie hätten also auch schreiben können:

```
VAR
AnfangszeileFenster,
AnfangsspalteFenster: Integer;
```

Der Typ Integer

Die Angabe des Typs Integer bedeutet, daß die Variable ganzzahlige Werte enthalten kann (englisch integer, „ganz", „ganzzahlig"). Zugelassen sind alle positiven ganzen Zahlen (1, 2, 3 ...), ebenso 0 oder negative ganze Zahlen (-1, -2, -3, -4 ...). Die anderen Grundtypen von Variablen werden etwas weiter unten besprochen.

Die Wertzuweisung

Im Anweisungsteil des Programms oder einer Prozedur kann eine Variable ganz ähnlich verwendet werden wie eine Konstante. Sie hat allerdings bei Programmbeginn noch keinen (beziehungsweise irgendeinen unsinnigen, undefinierten) Wert. Vor ihrer ersten Verwendung muß sie einen bestimmten Wert empfangen und kann – anders als eine Konstante – danach immer wieder neue Werte bekommen.

Die grundlegende Anweisung, die der Variablen ihren Wert gibt, heißt *Wertzuweisung*. In einer Wertzuweisung wird links der Variablenname hingeschrieben, gefolgt von Doppelpunkt und unmittelbar anschließendem Gleichheitszeichen. Rechts steht der Wert oder auch ein noch zu berechnender Ausdruck, auf den die Variable gesetzt werden soll. Die Computer interpretiert die Wertzuweisung von rechts nach links. Erst wird der Wert rechts gelesen oder ermittelt, dann das Ergebnis auf die Variable links gesetzt. Beispiele:

```
Anfangsspalte := 18;
Spalte        := Anfangsspalte + 3;
Anfangszeile  := 22;
Zeile         := Anfangszeile  - 4;
Hoehe         := Bodenzeile - Firstzeile;
```

Rechts in einer Wertzuweisung dürfen Variablen und Konstanten erscheinen, links kann nur eine Variable ste-

hen, die das Ergebnis aufnimmt. Alle beteiligten Variablen und Konstanten müssen selbstverständlich deklariert sein. Die Typen der Variablen auf der rechten Seite einer Wertzuweisung müssen entweder den gleichen Typ wie die Variable auf der linken Seite haben oder in diesen umwandelbar sein.

Die einfachen Variablentypen

In Standard-Pascal gibt es vier einfache Variablentypen, zu denen in Turbo-Pascal und anderen Implementationen weitere hinzukommen können. Den Reichtum an zusammengesetzten oder komplexen Datentypen, den Pascal und die Pascal-Abkömmlinge vor anderen Programmiersprachen auszeichnen, werden Sie später kennenlernen. Die vier einfachen Variablentypen in Pascal sind:

Integer *Integer* steht für die (negativen und positiven) ganzen Zahlen; Sie haben diesen Variablentyp schon kennengelernt. Das Unangenehmste an den Integer-Zahlen ist ihre Beschränkung auf einen begrenzten Bereich, in der Regel (und so auch in Turbo-Pascal) von -32768 bis +32767. Zahlen, die größer als der maximale, *MaxInt* genannte, Integer-Wert sind, gehören ebensowenig dazu wie weit im negativen Bereich liegende Zahlen. 32767 ist aber für viele Zwecke eine ziemlich kleine Zahl. Vorsicht müssen Sie auch bei Berechnungen walten lassen: 32767 + 1 ergibt zur großen Überraschung -32768. Die Integer-Zahlen sind sozusagen kreisförmig angeordnet. Das liegt an der Form, wie die Werte im Speicher dargestellt sind: Eine Integer-Zahl belegt im Speicher zwei Byte, 16 Bit, auf denen 16-stellige Dualzahlen, das heißt theoretisch bis 65536 stehen können. Aber die Hälfte dieser Zahlen wird für die negativen Zahlen verwendet (das Anfangsbit dient quasi als Vorzeichen).

Real *Real* heißt der Variablentyp für die Dezimalzahlen, die mit Dezimalpunkt geschrieben werden (zum Beispiel 233.41 oder 0.001). Die englischen reals sind im Deutschen die reellen Zahlen (so heißen in der Mathematik die Dezimalzahlen). Im Computer werden Dezimalzahlen in der sogenannten Gleitpunktdarstellung gespeichert, wie Sie es vielleicht schon am Taschenrechner gesehen haben. Eine Gleitpunktzahl (beziehungsweise Gleitkommazahl) besteht zum einen aus der Mantisse (in Turbo-Pascal elf-

stellig), einer Ziffernfolge, vorstellbar als eine mit „0" beginnende Dezimalzahl. Des weiteren besitzt sie einen Exponenten, der angibt, um wieviel Stellen das Komma (beziehungsweise der Dezimalpunkt) verschoben werden soll. Zum Beispiel sieht 233.41 in elfstelliger Gleitpunktdarstellung folgendermaßen aus:

0.23341000000E+3

„E" soll dabei anzeigen, wo die Exponentenzahl beginnt. In Turbo-Pascal wird intern der Exponent als 8-stellige Dualzahl (in einem Byte) und die Mantisse als 40-stellige Dualzahl (in fünf Bytes) dargestellt:

Wegen der Beschränkung des Integer-Bereiches muß notgedrungen bei großen Werten, die eigentlich ganzzahlig sein müssen, auch auf *Real* zurückgegriffen werden.

Ausgabeformat für Real-Größen

Wenn Sie nicht ausdrücklich etwas anderes programmieren, wird eine Real-Größe in der ziemlich unanschaulichen (aber den Speicherinhalt wiedergebenden) Gleitpunktdarstellung ausgegeben. In der Regel werden Sie aber mit einer normalen Dezimalzahl, zum Beispiel bei einem DM-Betrag mit zwei Stellen hinter dem Komma (oder hinter dem Punkt), arbeiten wollen. Dann müssen Sie das entsprechende Format bei der Ausgabe in der Write-Anweisung angeben. Dazu nennen Sie nach der auszugebenden Größe (zum Beispiel eine als Real vereinbarte Variable *Preis*):

- nach einem ersten Doppelpunkt die Gesamtbreite, in welcher die Dezimalzahl ausgerichtet werden soll,
- nach einem zweiten Doppelpunkt die Zahl der Stellen hinter dem Dezimalpunkt. Also zum Beispiel:.

```
Write (Preis :6:2, ' DM');
```

Anstelle der bestimmten Angaben wie 6 und 2 können Sie auch Integer-Konstanten oder Variablen verwenden. Falls Sie ein zu kleines Format für einen Wert angeben haben, wird das Format überschritten (und nicht etwa die Zahl abgeschnitten, was in anderen Programmiersprachen vorkommen kann). Verdeutlichen Sie sich das Gesagte über die Betrachtung der Bildschirmausgabe für die Werte 8.95, 98.50 und für den nicht mehr in das angegebene Format passenden Wert 5130.1233 anhand des folgenden Programmbeispiels:

```
PROGRAM Ausgabeformat;

VAR
  Preis: Real;

BEGIN
  Preis := 8.95;
  WriteLn (Preis :6:2, ' DM');
  Preis := 98.50;
  WriteLn (Preis :6:2, ' DM');
  Preis := 5130.1233;
  WriteLn (Preis :6:2, ' DM');
END.
```

Ausgabeformat für andere Typen

Übrigens können Sie die Formatierung auf eine Gesamtbreite auch bei anderen Typen von Ausgabegrößen anwenden. Setzen Sie einfach einen Doppelpunkt hinter die Variable oder den Ausdruck. Danach bezeichnen Sie die gewünschte Breite, auf welcher der Wert rechtsbündig erscheinen soll (Nachkommastellen gibt es dann natürlich nicht). Durch:

```
WriteLn (Name :40);
```

würde der Wert einer STRING-Variablen (vergleiche unten) *Name* rechtsbündig zur Bildschirmmitte gesetzt, wenn der Cursor vorher am Zeilenanfang stand. Ein Trick, um eine bestimmte Anzahl von Leerzeichen schreiben zu lassen, besteht darin, ein leeres Textstück mit beispielsweise *Anzahl* zu formatieren (wobei *Anzahl* eine Integer-Variable oder -Konstante sein muß):

```
WriteLn ('' :Anzahl);
```

Char

Char ist der spezielle Typ für einzelne Zeichen (englisch character, „Buchstabe, Zeichen"). Die Liste aller möglichen Char-Werte ist durch die Zeichensatztabelle des verwendeten Rechners definiert. Eine Char-Variable nimmt im Speicher ein Byte in Anspruch. Char-Werte werden stets in Apostrophen angegeben (so wie Sie es schon bei den Char-Konstanten am Ende des letzten Paragraphen sehen konnten):

```
'A'  'B'  'a'  'ß'  '0'  '1'  '2'  '♥'  '∞'
```

Boolean

Schließlich gibt es den Typ *Boolean* für die beiden logischen (booleschen) Werte „wahr" (true) und „falsch" (false). Für eine boolesche Variable würde an sich im Speicher nur ein einzelnes Bit benötigt. Da aber nur ganze

Bytes direkt ansprechbar sind, nimmt zum Beispiel Turbo auch jeweils ein Byte für einen booleschen Wert.

Etiketten mit Varianten

Mit wenig Aufwand können Sie die *Stempel mit Rahmen* aus dem letzten Kapitel, die Sie dort mit Hilfe einer Konstanten für das Rahmenzeichen kreierten, so umbauen, daß zwei oder mehr unterschiedliche Bilder gleichzeitig auf dem Bildschirm erscheinen. Sofern ihr System eine Hardcopy erlaubt, können Sie sich damit Aufkleber für unterschiedliche Zwecke herstellen.

Was ist dafür zu tun? Die vorher als Konstante vereinbarte Größe Z müssen Sie jetzt als Char-Variable deklarieren. Unter der Überschrift VAR steht im Deklarationsteil des Programms:

```
Anfangszeile  : Integer;
Anfangsspalte : Integer;
Z             : Char;    {RahmenZeichen}
```

Und vor dem Aufruf der Rahmenprozedur ist außer den jeweiligen Anfangszeilen und -spalten auch Z mittels

```
♥♥♥
```

einer Wertzuweisung mit dem richtigen Wert zu besetzen. Zum Beispiel für das Herz ♥ die Nummer #3; für „unendlich" der Char-Wert '∞' (beziehungsweise: #236) und so weiter. Im Hauptprogrammteil stünde dann:

```
BEGIN (* von GrosserStempel *)
  ClrScr;
  Z             :=   #3;
  Anfangszeile  :=    1;
  Anfangsspalte :=    1;
  Rahmen;
  Stempel;
  Z             := #236;
  Anfangsspalte :=   26;
  Rahmen;
  Stempel;
  Z             := #250;
  Anfangszeile  :=    8;
  Rahmen;
  Stempel;
  Z             := #240;
  Anfangsspalte :=    1;
  Rahmen;
  Stempel;
END.
```

So weit ist es ja ganz hübsch. Praktischer wäre aber, wenn sich auch der Text im Kasten variieren ließe. Sie benötigen hierzu Variablen, die nicht nur einzelne Zeichen, sondern ein ganzes Päckchen davon, zum Beispiel eine Reihe von fünfzehn einzelnen Zeichen aufnehmen können. Dafür gibt es in den meisten Pascal-Dialekten einen eigenen Typ.

Der Datentyp STRING

String ist im Englischen eine Schnur, an die Perlen aufgefädelt sein können. Ein String beim Programmieren bedeutet eine Kette von Zeichen. In Turbo-Pascal (und ähnlich bei den meisten Pascal-Arten, allerdings nicht in Standard-Pascal) gibt es einen vordefinierten Typ STRING für Variablen, die dazu bestimmt sind, Zeichenketten aufzunehmen. Um zum Beispiel die Prozedur *Stempel* universell zu halten, können Sie schreiben:

```
PROCEDURE Stempel;
  BEGIN
    GoToXY (Anfangsspalte + 4,
            Anfangszeile + 1);
    Write  (Name);
    GoToXY (Anfangsspalte + 4,
            Anfangszeile + 2);
    Write  (Strasse);
    GoToXY (Anfangsspalte + 4,
            Anfangszeile + 4);
    Write  (Ort);
  END (* von Stempel *);
```

STRING-Deklaration

Name, Strasse und *Ort* sind dabei STRING-Variablen, die wie alle Variablen vor ihrer Verwendung zu deklarieren sind. Bei der Deklaration ist außer dem Wort STRING noch in eckigen Klammern die Maximallänge, die die Zeichenkette haben soll, anzugeben. Wenn ab Turbo 4.0 diese Längenbegrenzung fehlt, gilt eine Maximallänge von 255. Also zum Beispiel:

```
Name:    STRING [18];
Strasse: STRING [18];
Ort:     STRING [18];
```

Alle Größen mit dem gleichen Typ dürfen bekanntlich auch gemeinsam vereinbart werden, das heißt, es darf auch geschrieben werden:

```
Name, Strasse, Ort: STRING [18];
```

An eine STRING-Variable werden im Ablauf des Programms STRING-Werte, also in Apostrophen eingerahmte Textstückchen, zugewiesen.

```
Name := 'Rupert Röder';
Stempel;
Name := 'Michaela Meyer';
Stempel;
```

Ist ein Textstück länger als maximal vorgesehen, so wird der Überschuß in der Regel einfach wegfallen. In Turbo-Pascal kann ein STRING maximal 255 Zeichen umfassen.

Aufgaben

1. Bauen Sie Ihr Haus vom letzten Kapitel zu einer Villa mit vielen Fenstern und Türen aus oder erzeugen Sie aus vielen Häusern eine ganze Straße. Sie können das Aussehen der Häuser mit Hilfe von Char- und STRING-Größen variieren.

2. Wenn Sie über einen Drucker verfügen, schreiben Sie sich ein Programm zum Druck von Einmach-Etiketten, das jeweils drei Etiketten nebeneinander und mehrere Reihen untereinander erstellt. Dabei sollen die Etiketten – von einer konstanten Anfangsnummer ausgehend – fortlaufend numeriert sein.

3. Schreiben Sie ein Programm, das die Einladungskarten für eine Feier nach folgendem Muster schreibt (und, wenn möglich, je drei auf den Bildschirm bringt). Überlegen Sie sich erst, wie das Strukturdiagramm aussieht!

```
                                                  ♥ ♣
                                Mau-Mau-Club      ♦ ♠
                                       Mainz
Liebe ...............
wir laden herzlich zu unserer Feier am 21.6. um 18 h in
den bekannten Räumen ein. Getränke und ................
bitte mitbringen.


Dein
```

Abbildung 4-3: Muster für Einladungskarten

5 Die FOR-Anweisung

Ein erstes Beispiel zur FOR-Anweisung – Zählschleifen – Allgemeine Beschreibung – Syntaxdiagramme – Verbundanweisungen – Zerlegung von STRINGs – Aufgaben

Das wesentliche Element, durch das ein Programm scheinbar zu eigenem Leben erweckt wird, ist die gesteuerte Wiederholung von Teilabläufen. Als ein erstes Hilfsmittel, um so etwas zu bewirken, lernen Sie die FOR-Anweisung kennen. Dabei können Sie gleichzeitig auch einen genaueren Blick auf die im letzten Kapitel vorgestellten Datentypen werfen.

Ein erstes Beispiel zur FOR-Anweisung

Betrachten Sie noch einmal das Beispiel *GrosserStempel* in der Fassung, bei der Konstanten für Textzeichen verwendet wurden. Mittels einer Char-Variablen Z gab das Programm den Stempel mit verschiedenen Umrahmungen auf dem Schirm aus (*PROGRAM GrosserStempel; {In perfekter Form!}*). Möchten Sie ohne viel Aufwand alle möglichen Rahmenzeichen durchgehen und anschauen, wie das Bild jeweils wirkt? Wandeln Sie dazu das Programm wie folgt ab:

```
PROGRAM TesteRahmenzeichen
                          {Variante des GrossenStempels};

USES crt, printer;

CONST
  Anfangszeile   =  8;
  Anfangsspalte = 20;

VAR
  Z : Char;
```

```
PROCEDURE Stempel;
  BEGIN
    GoToXY (Anfangsspalte + 4, Anfangszeile + 1);
    Write  ('Rupert Röder');
    GoToXY (Anfangsspalte + 4, Anfangszeile + 2);
    Write  ('Pascalweg 123');
    GoToXY (Anfangsspalte + 4, Anfangszeile + 4);
    Write  ('65oo Mainz');
  END (* von Stempel *);

PROCEDURE Rahmen;
  BEGIN
    GoToXY (Anfangsspalte, Anfangszeile + 0);
    Write  (Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z);
    GoToXY (Anfangsspalte, Anfangszeile + 1);
    Write  (Z,'                    ',Z);
    GoToXY (Anfangsspalte, Anfangszeile + 2);
    Write  (Z,'                    ',Z);
    GoToXY (Anfangsspalte, Anfangszeile + 3);
    Write  (Z,'                    ',Z);
    GoToXY (Anfangsspalte, Anfangszeile + 4);
    Write  (Z,'                    ',Z);
    GoToXY (Anfangsspalte, Anfangszeile + 5);
    Write  (Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z,Z);
  END (* von Rahmen *);

BEGIN (* von Hauptprogramm *)
  FOR Z := '!' TO '■'DO          {Oder, was dasselbe ist: FOR Z
                                  := #33 TO #254}
    BEGIN
      ClrScr;
      Rahmen;
      Stempel;
      GoToXY (3,20);
      Write  ('Drücken Sie bitte die Taste RETURN, ');
      Write  ('um verschiedene Rahmen anzuschauen!');
      ReadLn;
    END;
END.
```

Schema der FOR-Anweisung

Das Schema der FOR-Anweisung läßt sich so angeben:

```
FOR Z := Anfangswert TO Endwert DO
  BEGIN
    { ... }
  END;
```

In Worten ausgedrückt, bedeutet dies:

1. Setze auf die Variable Z den Anfangswert (im Beispiel '!').

2. Führe aus, was hinter DO steht. Dabei (ganz wichtig!) gehören alle Anweisungen zu DO, die durch BEGIN-END umrahmt sind.

3. Die Schritte 1. und 2. führe ebenso durch für den nach Anfangswert nächsten Wert, danach für den Wert, der auf diesen folgt, und so weiter, bis sie schließlich für den Endwert vollzogen sind.

Da der dritte Schritt das Programm anweist, mehrfach an einer früheren Stelle wieder anzusetzen, eine Reihe von Anweisungen kreisförmig zu durchlaufen, wird von einer FOR- oder auch Lauf*schleife* gesprochen.

Anwendung auf das Beispiel

Das Beispielprogramm führt also die Anweisungen, die hinter DO zwischen BEGIN und END stehen, für jedes Zeichen ab dem Ausrufezeichen bis zum einschließlich letzten Zeichen der Zeichentabelle aus. Jeweils wird zunächst der Bildschirm gelöscht, dann der Stempeltext geschrieben und der Rahmen gemalt. Der Rahmen wird dabei mit dem jeweilig vorhandenen Inhalt der Rahmenzeichenvariable Z auf den Bildschirm ausgegeben (also erst mit einem Ausrufezeichen, dann mit Gänsefüßchen und so weiter).

ReadLn

Die Anweisung ReadLn (Abkürzung für read line, also „lies Zeile") ist auch noch zu erklären: Sie bewirkt, daß der Programmablauf an dieser Stelle anhält, bis die RETURN- beziehungsweise ENTER-Taste gedrückt wird. Dadurch können Sie entscheiden, wie lange Sie einen Rahmen anschauen wollen.

Zählschleifen

Im Beispiel wurde durch die FOR-Anweisung ein Teil des Zeichensatzes durchgegangen. Mit diesen Char-Werten wurde dann eine Bildschirmausgabe aktiviert. Ähnlich können auch Abschnitte des Typs Integer durchlaufen werden. Der Wert der Integer-Variable, die dann an der Stelle von Z die FOR-Schleife steuert, erhöht sich dabei immer um 1. Im einfachsten Fall beginnt die abgeklapperte Zahlenfolge mit 1 und endet mit einer festen Zahl. Die Schleifenvariable zählt hier praktisch, wie oft die Anweisungen ausgeführt werden. Die Schleife heißt daher

auch Zählschleife. Das folgende Beispiel, Movie1 genannt, illustriert diese Verwendung der FOR-Anweisung.

Delay Damit das Programm effektvoller abspult, ist eine Verzögerungsanweisung eingebaut: In Turbo-Pascal kann die interne Prozedur Delay (delay, „Verzögerung") aufgerufen werden, die das Programm eine bestimmte Zahl von Millisekunden anhält (die Zahl wird in Klammern beigefügt).

```
PROGRAM Movie1;

USES crt, printer;

CONST
  Verzoegerungswert = 300;

VAR
  Zahl: Integer;

PROCEDURE PfeilMalen;
  BEGIN
    Write ('*--> ');
  END (* von PfeilMalen *);

PROCEDURE KanteMalen;
  BEGIN
    Write ('o+++- ');
  END (* von KanteMalen *);

BEGIN (* von Hauptprogramm *)
  ClrScr;
  FOR Zahl := 1 TO 30 DO
    BEGIN
      PfeilMalen;
      Delay (Verzoegerungswert);
    END;
  WriteLn;
  WriteLn; Delay (1000);
  FOR Zahl := 1 TO 30 DO
    BEGIN
      KanteMalen;
      Delay (Verzoegerungswert);
    END;
END.
```

Wenn Sie eine Pascal-Variante benutzen, die nicht über eine Delay-Anweisung verfügt, können Sie eine zusätzliche FOR-Schleife anwenden. Diese macht nichts anderes,

als dem Computer etwas Rechenarbeit zu bereiten, so daß der Programmablauf einen Moment stockt:

```
FOR Wartezahl := 1 TO 1000 DO
  BEGIN
  END;
```

Wartezahl muß als Integer-Variable deklariert sein. Experimentieren Sie mit unterschiedlichen *Verzoegerungswerten* und verschiedenen Endwerten für die Laufschleife. Vielleicht möchten Sie auch die Prozeduren *PfeilMalen* und *KanteMalen* weiter ausgestalten. Mit einer geringfügigen Erweiterung wird das Programm zu Movie2. Deklarieren Sie noch die folgende Prozedur, die so viele Leerzeichen auf den Schirm schreibt, wie die Variable *Zahl* vorgibt:

```
PROCEDURE VerschiebungUmZahl;
  VAR
    Nummer: Integer;
  BEGIN
    FOR Nummer := 1 TO Zahl DO
    BEGIN
      Write (' ');
    END;
  END (* von VerschiebungUmZahl *);
```

Rufen Sie dann die Prozedur jeweils vor *PfeilMalen* beziehungsweise *KanteMalen* auf, und setzen Sie zwischen *PfeilMalen* beziehungsweise *KanteMalen* und dem Delay ein WriteLn. Das Programm wird einen ziemlich veränderten Effekt zeigen. Als weitere Veränderung des Programms sollten Sie noch ausprobieren, die gemalten Pfeile und Kanten auf dem Bildschirm mitzählen zu lassen. Dazu fügen Sie die folgende Programmzeile noch in die Anweisungen, die zum DO gehören, ein:

```
Write (Zahl);
```

Allgemeine Beschreibung

Sie haben jetzt mehrere Beispiele mit FOR-Anweisungen gesehen. Was Sie dabei schon praktiziert haben, sei nochmals festgehalten: In der FOR-Anweisung wird eine (Kontroll- oder Steuervariable genannte) Größe auf einen Anfangswert gebracht und anschließend jeweils um ei-

nen Wert weitergesetzt, solange ein Endwert noch nicht erreicht ist. Es darf vorkommen, daß der Endwert vor dem Anfangswert liegt (das passiert leicht, wenn die Werte nicht direkt, sondern als Rechenausdrücke angegeben sind). In diesem Fall werden die zum DO gehörenden Anweisungen gar nicht ausgeführt, im anderen Fall für jeden Wert der Kontrollvariable einmal. Die Kontrollvariable kann in diesen Anweisungen für unterschiedlichste Zwecke verwendet werden, soll dabei aber keinesfalls einen neuen Wert erhalten.

Beachten Sie: Die Typen von Kontrollvariable, Anfangswert und Endwert müssen zueinander passen. Die Werte vom Anfangs- bis zum Endwert sollen ja auf die Kontrollvariable gebracht werden. Ist die Kontrollvariable eine Integer-Zahl, so müssen es auch Anfangs- und Endwert sein. Das Entsprechende gilt für Zeichen.

Nur Aufzählungstypen in der FOR-Anweisung

Real- und STRING-Variablen können nicht als Kontrollvariablen benutzt werden (und daher auch Real- und STRING-Werte nicht als Anfangs- oder Endwerte auftauchen). Denn bei ihnen gibt es keine klare Abfolge von Werten, die zu durchlaufen wäre. Bei Integer-Zahlen ist eine natürliche Folge durch das Wachsen in Einerschritten gegeben. Bei Zeichen wird die Reihenfolge der Zeichensatztabelle des Rechners zugrunde gelegt. Später werden noch weitere Typen vorgestellt, deren Werte ebenfalls per Definition eine bestimmte Reihenfolge haben und dann auch mit einer FOR-Schleife durchlaufen werden können. Allgemein heißt ein Typ ein Aufzählungstyp, wenn für seine Werte eine festgelegte Reihenfolge, eine Liste oder eine Aufzählung existiert. Integer und Char sind also Aufzählungstypen, Boolean formal auch, Real und STRING sind keine.

Die Anweisungen, die zum DO-Teil der FOR-Anweisung gehören, werden durch BEGIN-END umrahmt. Diese BEGIN-END-Klammer darf entfallen, wenn nur eine einzelne Anweisung in jedem Schleifendurchgang abzuarbeiten ist. Die Prozedur *VerschiebungUmZahl* könnten Sie also auch wie folgt schreiben:

```
PROCEDURE VerschiebungUmZahl;
VAR
  Nummer: Integer;
BEGIN
  FOR Nummer := 1 TO Zahl DO
```

```
    Write ('  ');
END (* von VerschiebungUmZahl *);
```

Die FOR-Anweisung schließt hier mit dem Ende der ihr zugeordneten, von ihr abhängigen Write-Anweisung.

Lokale Kontrollvariable

Wichtig ist noch: In Turbo-Pascal ab Version 4.0 sowie in Quick-Pascal muß die Schleifenkontrollvariable global für das gesamte Programm oder lokal in der jeweiligen Prozedur vereinbart sein. Wegen ihrer rein lokalen Bedeutung sollte sie in der Regel lokal definiert werden.

Syntaxdiagramme

Ein kurzes Beispiel ist oft viel besser zu verstehen als die genaue theoretische Beschreibung des Aufbaus einer Anweisung. Beispiele haben allerdings den Nachteil, daß nicht alle möglichen Fälle durch sie abgedeckt werden. Es muß zusätzlich eine allgemeine Beschreibung geben, bei der Sie im Zweifelsfalle nachschauen können. Außerdem muß bei der Entwicklung der Sprache exakt festgelegt werden, was erlaubt sein soll und was nicht. In manchen Programmiersprachen ist es, ähnlich wie in diesem Buch, üblich, diese allgemeine Beschreibung mehr oder weniger umgangssprachlich zu fassen. Manchmal würde eine solche Beschreibung aber lang und unübersichtlich werden. Unübersichtlichkeit ist der Todfeind eines sicheren Programmierens, erst recht des Entwurfs einer zuverlässigen Programmiersprache.

Syntax

Die Informatik hat deshalb noch andere Methoden zur knappen und dennoch genauen Beschreibung der zulässigen Ausdrücke einer Programmiersprache entwickelt. Die in der Informatik wichtigste Methode ist das Syntaxdiagramm; so genannt, weil mit dem Fachbegriff Syntax – nicht nur bei Computersprachen – die Regeln bezeichnet werden, wie aus Sprachelementen größere Ausdrücke gebildet werden dürfen.

Bestandteile des Syntaxdiagramms

Ein Syntaxdiagramm ist eine Zeichnung, die aus Ovalen für feste Elemente der Sprache, aus rechteckigen Kästchen für zu definierende Ausdrücke und aus gerichteten Verbindungslinien zwischen Ovalen und Kästchen besteht. Wenn man entlang den Verbindungslinien fährt, erkennt man, wie die Sprachbestandteile aneinandergehängt werden dürfen. Alles, was in Ovalen steht, ist genau so auch in das Programm zu schreiben, die Bedeu-

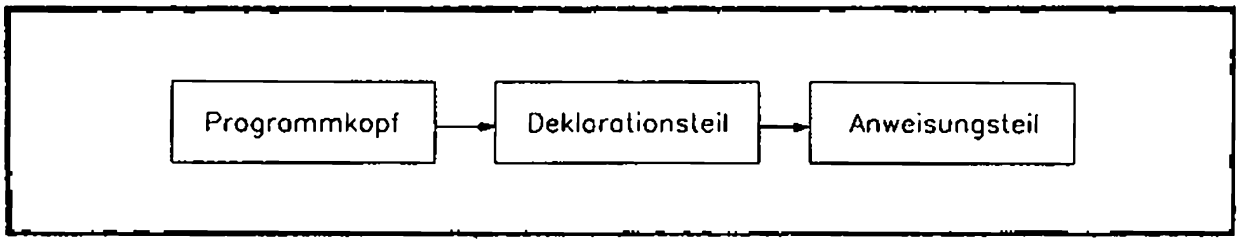

Abbildung 5-1: Syntaxdiagramm „Pascal-Programm"

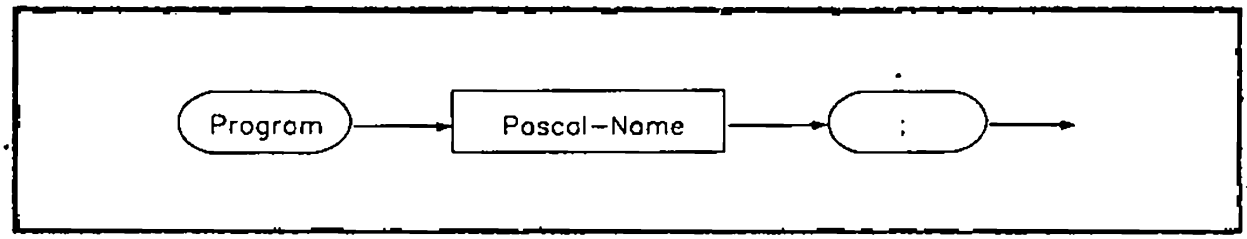

Abbildung 5-2: Syntaxdiagramm „Programmkopf"

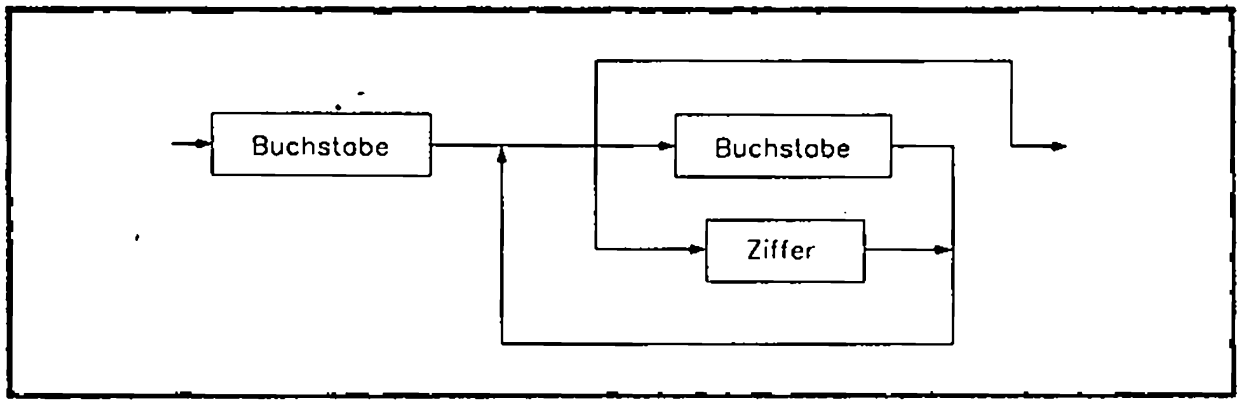

Abbildung 5-3: Syntaxdiagramm „Pascal-Name"

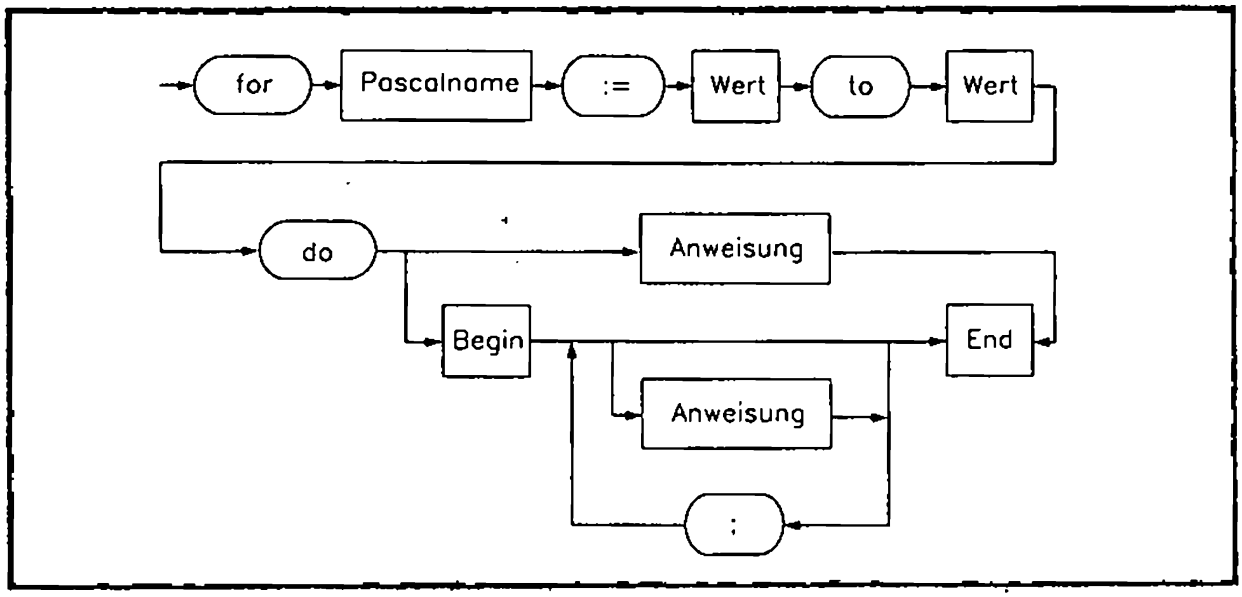

Abbildung 5-4: Syntaxdiagramm „FOR-Anweisung"

tung der Inhalte der rechteckigen Kästchen muß noch zusätzlich erklärt werden. Sie verstehen sicherlich sofort das Diagramm für den prinzipiellen Aufbau eines Pascal-Programms (Abb. 5-1, S. 64).

Was der Programmkopf ist, kann dann mit einem weiterem Diagramm beschrieben werden (Abb. 5-2, S. 64).

Hier bleibt noch Pascal-Name zu erklären, wofür sich wieder ein Syntaxdiagramm eignet (Abb. 5-3, S. 64).

Dieses Diagramm drückt aus, daß ein Pascal-Name mit einem Buchstaben zu beginnen hat, und dann eine beliebige Folge von Buchstaben und Ziffern kommt. Fahren Sie seinen Linien nach, als wäre es ein Eisenbahn-Streckennetz! Sie werden beobachten, daß ein solches Diagramm eine mindestens so präzise Beschreibung darstellt wie ein deutscher Satz, bei dem man nicht immer genau weiß, was zum Beispiel „Folge" heißen soll. Wie in Pascal der Programmkopf aufgebaut ist, könnte nun leicht bis auf die elementaren Bausteine zurückgeführt werden. Es fehlt nur noch die Festlegung, was alles ein Buchstabe beziehungsweise eine Ziffer sein kann. Das Syntaxdiagramm der FOR-Anweisung zeigt Abbildung 5-4, S. 64.

Verbundanweisungen

Wie Sie bei der FOR-Anweisung sehen, umschließt ein BEGIN-END-Rahmen nicht immer nur den Anweisungsteil des Hauptprogramms oder einer Prozedur. Mit ihm werden auch Anweisungen zu einer Einheit verschnürt, um beispielsweise auszudrücken, daß die gebündelte Anweisungsreihe insgesamt zum DO gehört. Eine durch BEGIN-END geklammerte Anweisungsreihe ist praktisch zu einer einzelnen Anweisung höherer Ebene zusammengewachsen und trägt dann auch den Namen Verbundanweisung.

Wie schon erwähnt, hat Pascal keine zeilenorientierte Struktur. Grundsätzlich ist die Anordnung der Elemente eines Programms dem/der Programmierenden überlassen.

Empfehlung

Um so mehr sollte man sich Mühe geben, sich ein klares Schema zur Schreibweise von Programmen zu überlegen und sich halbwegs konsequent daran zu halten, damit der Text möglichst übersichtlich bleibt. Das wird bei der FOR-Anweisung zum ersten Mal wirklich wichtig. Eine Möglichkeit ist, alles, was im selben Abhängigkeits-

grad von einem übergeordneten Element steht, auf derselben senkrechten Linie beginnen zu lassen:

```
PROCEDURE VerschiebungUmZahl;

VAR
          {VAR leitet den Deklarationsteil
           für Variablen ein.}
  Nummer: Integer;
          {Nummer ist ein Teil der VAR-
           Deklarationen, sollte also
           weiter eingerückt sein.}

BEGIN
          {Hier fängt der Anweisungsteil
           an.}
  FOR Nummer := 1 TO Zahl DO
          {Die erste Anweisung; das DO und
           das folgende BEGIN leiten die
           Anweisungen ein, die den Rumpf
           der FOR-Anweisung bilden.}

    BEGIN
      Write (' ');
          {Das ist eine Anweisung im Rumpf
           der FOR-Anweisung.}
    END;
END (* von VerschiebungUmZahl *);
```

Sehen Sie sich unter diesem Gesichtspunkt einmal die längeren Programmbeispiele an.

Umkehrung der Durchlaufrichtung

Es gibt auch die Möglichkeit, in der FOR-Anweisung die Werte in umgekehrter Reihenfolge – von größeren zu kleineren – vorzugeben. Dafür müssen Sie TO durch DOWNTO (ungefähr „abwärts zu") ersetzen und sinnvollerweise für den Anfangswert einen größeren Wert als für den Endwert nehmen. Zum Beispiel zählt die Anweisung (mit einer Integer-Variablen *Zahl*):

```
FOR Zahl := 1000 DOWNTO 1 DO
  WriteLn (Zahl);
```

die Zahlen von 1 bis 1000 rückwärts, oder:

```
FOR Zeichen := 'z' DOWNTO 'a' DO
  Write (Zeichen);
```

buchstabiert das kleine Alphabet von hinten durch.

Zerlegung von STRINGs

Zum Schluß diese Kapitels noch eine Anwendung: Eine STRING-Größe enthält eine Zeichenkette. Die einzelnen Zeichen können direkt angesprochen werden durch den Namen der STRING-Variablen, gefolgt von der Nummer des Zeichens in eckigen Klammern. Ist zum Beispiel *Name* eine STRING-Variable und hat es eine Wertzuweisung gegeben:

```
Name := 'Rupert Röder'
```

so ist

```
Name [1]
```

das erste Zeichen des STRINGs *'Rupert Röder'*, also *'R'*; *Name [2]* steht entsprechend für *'u'* und so weiter. Die Zeichennummer kann auch von einer Variablen repräsentiert werden. Diese Variable kann zum Beispiel die Nummern vom ersten bis zum letzten Zeichen durchlaufen.

Length ()

Damit dies geht, müssen Sie allerdings wissen, welche Nummer das letzte Zeichen eines STRINGs hat. Dafür gibt es in Turbo-Pascal einen eigenen Ausdruck: *Length (Name)*. Length gibt die Länge des STRINGs *Name* an, das heißt, Length enthält die Zahlenangabe, aus wieviel Zeichen der STRING besteht. Length (Name) läßt sich deshalb wie eine Zahl verwenden. Zum Beispiel würde die Anweisung:

```
Write (Length (Name))
```

die Zahl der Zeichen von *Name* auf dem Bildschirm ausgeben. Und so können Sie *Length (Name)* auch als Endwert einer FOR-Anweisung benutzen. Dadurch erreichen Sie, daß alle Zeichen eines STRINGs vom ersten bis zum letzten erfaßt werden. Wenn *i* eine Integer-Variable ist, so bewirkt:

```
FOR i := 1 TO Length (Name) DO
  Write (Name [i]);
```

daß nacheinander das erste, das zweite ... (allgemein: das i-te) Zeichen bis zum letzten Zeichen des STRINGs *Name* auf dem Bildschirm erscheint (es wird in diesem Zusammenhang von einem Index gesprochen und daher gern die Abkürzung *i* gewählt). Die Anweisung bedeutet also genau das gleiche wie:

```
Write (Name)
```

Wenn man den STRING auflöst, kann man ihn auch verändern, zum Beispiel ihn g e s p e r r t oder g-e-s-t-r-i-c-h-e-l-t ausgeben:

```
PROGRAM Gesperrt_Gestrichelt;

USES crt, printer;

VAR
  Name : STRING;
  i    : Integer;

BEGIN (* von Hauptprogramm *)
  Name := 'Rupert Röder';
  ClrScr;
  FOR i := 1 TO Length (Name) DO
    Write (Name [i], ' ');
  GoToXY (1,2);
  Write (Name [1]);
  FOR i := 2 TO Length (Name) DO
    Write ('-', Name [i]);
END.
```

Können Sie sich erklären, warum in der letzten FOR-Anweisung die Ausgabe des ersten Buchstabens herausgenommen werden muß?

Aufgaben

1. Schauen Sie sich Ihre bisher geschriebenen Programme noch einmal daraufhin an, ob Sie den Programmtext übersichtlich und entsprechend der logischen Gliederung eingerückt haben. Korrigieren Sie eventuell.

2. Schreiben sie ein Programm, das mittels einer Zählschleife eine größere Zahl von identischen Adreßaufklebern oder Visitenkarten auf dem Bildschirm ausgibt.

```
                /————————————$   Gleis a
————————————————/————————————————— Gleis b
——*————————/———————————————————— Gleis c
```

Abbildung 5-5: Gleisplan des Kleinbahnhofs Tuttlingen

3. Fügen Sie eine oder mehrere Schwalben (zum Beispiel mit der geschweiften Klammer darzustellen) zu Ihrem Haus hinzu und lassen Sie sie langsam am Himmel entlangziehen.

4. Und jetzt ein Leckerbissen für die Freunde und Freundinnen der Eisenbahn und des Reisens: Sie sehen hier (Abb. 5-5) den Gleisplan des Kleinbahnhofs Tuttlingen.

Es ist nicht schwierig, ihn auf den Bildschirm zu bringen. Von links kommt auf Gleis c gerade eine Lok angefahren, symbolisiert durch das Sternchen. Lassen Sie sie auf dem Bildschirm über das Gleis nach rechts wandern. Verwenden Sie die Delay-Anweisung, damit die Bewegung effektvoller vonstatten geht.

5. Schreiben Sie ein Programm, das einen STRING g*e*s*t*e*r*n*t ausgibt.

5.1 Schreiben Sie ein Programm, das einen STRING folgendermaßen schreibt:

c
h
i
n
e
s
i
s
c
h

5.2 Schreiben Sie ein Programm, das einen STRING sträwkcür (rückwärts) zeigt.

6 Dialogprogramme

Das Palindrom-Beispiel – Der Eingabedialog – Masken – Das Programmierschema E-V-A – Ausgabe auf dem Drucker – Aufgaben

„Hallo, hier ist Ihr Computer und tut, als ob" – Bei vielen Computeranwendungen scheint das Gerät mit dem Benutzer oder der Benutzerin zu sprechen. „Er" stellt Fragen, erwartet eine Aussage, reagiert störrisch, wenn man etwas Unpassendes macht, antwortet ansonsten mehr oder weniger entsprechend der Eingabe. Ein Programm, das den Computer zu diesem Verhalten bringt, wird ein Dialogprogramm genannt.

Das Palindrom-Beispiel

Palindrom heißt ein Wort oder ein Satz, das oder der vorwärts wie rückwärts gelesen gleich ist. Zum Beispiel sind „Anna" und „Otto" einfache Palindrome. Entsprechende Palindromsätze zu bauen, ist schon schwieriger. Der bekannteste Palindromsatz lautet: „Ein Neger mit Gazelle zagt im Regen nie." Bei der Umkehrung müssen die Leerstellen zwischen den Wörtern und die Groß- und Kleinschreibung neu gesetzt werden. Es wird auch von Sinn-Palindromen gesprochen, wenn die Umkehrung in einem inhaltlichen Bezug zum Originalwort steht: Roma – Amor könnte ein Beispiel sein. Ein Programm soll dabei helfen, Wörter oder längere STRINGs auf ihre Palindromeigenschaften zu testen.

Programmierung mit den bisherigen Mitteln

Eine Prozedur *Umkehrung* ist mit den Mitteln des letzten Kapitels schnell geschrieben. Dabei soll *Textstueck* eine Variable des Hauptprogramms sein, die den rückwärts auszugebenden STRING enthält. Ein Programm, das die Prozedur verwendet, könnte etwa wie folgt aussehen:

```
PROGRAM PalindromTest;

USES crt, printer; VAR
  Textstueck: STRING [200];

PROCEDURE Umkehrung;
  VAR
```

```
    i: Integer;
  BEGIN
    FOR i := Length (Textstueck) DOWNTO 1
    DO
      BEGIN
        Write (Textstueck [i]);
      END;
  END (* von Procedure-Umkehrung *);

BEGIN (* von Hauptprogramm *)
  ClrScr;
  Textstueck := 'Reliefpfeiler';
  WriteLn ('Vorwärts:');
  WriteLn (Textstueck);
  WriteLn;
  WriteLn ('Rückwärts:');
  Umkehrung;
  ReadLn;
END.
```

In dieser Art wurden bisher alle Programme geschrieben, aber hier wird die Starrheit des Programms unpraktisch. Für jedes neue Wort, das getestet werden soll, muß im Programmtext die Wertzuweisung an *Textstueck* geändert werden. Viel hübscher wäre, wenn das Programm fest bleiben und man beim Ablauf erst bestimmen könnte, welchen Wert die Variablen annehmen sollen. Genau dafür gibt es eine eigene Sorte von Anweisungen: die Eingabeanweisungen. In Pascal sind das die Anweisungen *Read* beziehungsweise *ReadLn*.

Funktionsweise von Read und ReadLn

Die Pascal-Wörter Read beziehungsweise ReadLn (read line) stehen für „Lesen" beziehungsweise „Lesen bis Zeilenende". Gemeint ist das Folgende: Bei einer Read- oder ReadLn-Anweisung stoppt der Programmablauf, bis der Computer von der Tastatur als Eingabegerät irgendetwas „gelesen" und es in seinen Speicher genommen hat. Das Gelesene schreibt der Computer dabei in eine Variable, die in runden Klammern hinter Read beziehungsweise ReadLn stehen muß. Zum Beispiel kommen durch:

```
ReadLn (Textstueck)
```

die eingetippten Zeichen von der Tastatur in die Variable *Textstueck*. Genauso können auch Zahlen-Variablen eingelesen werden. Und mit der folgenden Programmzeile wird eine eingegebene Zahl in die Variable *Nummer* übertragen und steht dann dort zur Verfügung:

```
ReadLn (Nummer)
```

Fehlerhafte Zahleneingaben

Passen Sie auf, daß Sie sich nicht vertippen: Ist *Nummer* als Integer deklariert, so stürzt das Programm mit einer Fehlermeldung ab, wenn etwas anderes als eine ganze Zahl eingetippt wird. Besonders aufpassen müssen Sie bei Real-Größen, weil eine Dezimalzahl nur mit Dezimalpunkt, nicht etwa mit Komma, geschrieben werden darf.

Unterschied von Read und ReadLn

In Turbo-Pascal sind in der einfachsten Benutzungsart alle Eingaben, sowohl bei Read, als auch bei ReadLn mit der Return-Taste abzuschließen. Der Unterschied zwischen den Anweisungen besteht darin, daß bei ReadLn auf dem Bildschirm nach dem Ende der Eingabe in die nächste Zeile gesprungen wird, bei Read nicht. Der Sonderfall einer alleinstehenden ReadLn-Anweisung ohne Variable wurde schon erwähnt: Dabei stoppt das Programm solange, bis Return gedrückt wird.

Der Eingabedialog

Die Anweisung

```
ReadLn (Textstueck);
```

allein ergibt noch kein vernünftiges Programm, lediglich der Cursor blinkt, ohne daß eine konkrete Eingabeanforderung erscheint. Grundsätzlich sollte das Programm mitteilen, welche (Art von) Eingabe erwartet wird. Dafür eignet sich ein kleiner Dialog der oben angedeuteten Art. Die Prozedur *Eingabedialog* wird folgendermaßen geschrieben:

```
PROCEDURE Eingabedialog;
  BEGIN
    WriteLn;
    WriteLn ('Das Programm Palindrom hilft, STRINGs auf',
             'Palindromeigenschaften zu testen.');
    WriteLn ('::::::::::::::::::::::::::::::::::::::::::',
             '::::::::::::::::::::::::::::::::::');
```

```
    WriteLn;
    WriteLn ('Geben Sie bitte das zu testende Wort ein:');
    ReadLn (Textstueck);
    WriteLn;
    WriteLn;
  END (* von Eingabedialog *);
```

Schreiben Sie diese Prozedur in das Programm *PalindromTest*, und ersetzen Sie die Wertzuweisung *Textstueck* durch einen Aufruf von *Eingabedialog*. Der Hauptprogrammteil sieht dann folgendermaßen aus:

```
BEGIN (* Hauptprogrammteil *)
  ClrScr;
  Eingabedialog;
  WriteLn ('Vorwärts:');
  WriteLn (Textstueck);
  WriteLn;
  WriteLn ('Rückwärts:');
  Umkehrung;
END.
```

Jetzt haben Sie ein praktisches Programm, mit dem Sie experimentieren können, in dem Sie es einfach immer wieder neu ablaufen lassen.

Masken

Wenn ein Programm eine Serie etwas komplizierterer Eingaben benötigt, kann es sich empfehlen, eine Bildschirmmaske zu programmieren. In der einfachsten Form heißt dies, daß die Reihe der Fragen, die der Computer zu einer Aufgabe stellen soll, nicht nacheinander auftauchen, sondern nach Art eines Formulars gleichzeitig auf den Bildschirm kommt. Bei der Beantwortung läßt sich dadurch besser Zusammenhang und Sinn der Eingabe erkennen.

Beispiel „Namensschildchen"

Betrachten Sie als Beispiel eine Abwandlung des *Stempels*: Ein Namensschildchen mit Angabe des Herkunftsortes (die Prozedur *Eingaben*, in der eine Maske zur Datenerfassung verwendet wird, ist noch ausgeklammert und wird unten nachgeliefert):

```
PROGRAM Namensschildchen;

USES crt, printer;

CONST
  Anfangszeile  =  8;
  Anfangsspalte = 20;

VAR
  Vorname, Name, Ort          : STRING [80];
  GesamtlaengeUmrahmterText : Integer;

PROCEDURE Eingaben;        {Eingabemaske für Vorname, Name und Ort}
  BEGIN
                           { ... }
  END (* von Eingaben *);

PROCEDURE StrichMalen;
  VAR
    i: Integer;
  BEGIN
    FOR i := 1 TO GesamtlaengeUmrahmterText DO
      Write ('-');
  END (* von StrichMalen *);

PROCEDURE SchildMalen;
  BEGIN (* von SchildMalen *)
    GesamtlaengeUmrahmterText
    := Length (Vorname) + 1 + Length (Name) + 2 + Length (Ort);
    GoToXY (Anfangsspalte, Anfangszeile);
    Write  ('┌──');
    StrichMalen;
    Write  ('──┐');
    GoToXY (Anfangsspalte, Anfangszeile + 1);
    Write  ('│  ');
    Write  (Vorname, ' ', Name, ', ', Ort);
    Write  ('  │');
    GoToXY ( Anfangsspalte,  Anfangszeile + 2) ;
    Write  ('└──');
    StrichMalen;
    Write  ('──┘');
  END (* von SchildMalen *);

BEGIN (* Namensschildchen *)
  ClrScr;
  Eingaben;
  SchildMalen;
END.
```

Das Programm setzt einen Rahmen um den Namen und die Ortsangabe entsprechend den jeweiligen Längen dieser Angaben. Bevor es funktioniert, muß man die Prozedur *Eingaben* ausfüllen.

Eine Maskenprozedur

Die Eingabe von Namen und Orten sollen in einer Maske in einfacher Form erfolgen. Dazu muß man sich zunächst überlegen, wie der Bildschirm (das heißt sozusagen das Eingabeformular) aufgebaut sein soll, etwa so:

```
Angaben für Namensschildchen
============================
Vorname      :
Nachname     :
Herkunftsort:
```

Abbildung 6-1: Maske zur Datenerfassung für das Programm Namensschildchen

Mit Hilfe von GoToXY-Anweisungen ist zunächst diese Maske auf dem Bildschirm darzustellen. Anschließend werden die einzelnen Positionen, an denen die Eingaben geschrieben werden, angesprungen. Das kann zum Beispiel so aussehen:

```
PROCEDURE Eingaben;
  CONST
   Anfangszeile   =  2;
   Anfangsspalte  =  4;
   Textbreite     = 12;
  BEGIN
    ClrScr;
    GoToXY (Anfangsspalte, Anfangszeile);
    Write ('Angaben für Namensschildchen');
    GoToXY (Anfangsspalte, Anfangszeile+1);
    Write ('============================');
    GoToXY (Anfangsspalte, Anfangszeile+3);
    Write ('Vorname');
    GoToXY (Anfangsspalte, Anfangszeile+4);
    Write ('Nachname');
    GoToXY (Anfangsspalte, Anfangszeile+5);
    Write ('Herkunftsort');
    GoToXY (Anfangsspalte+Textbreite, Anfangszeile+3);
```

```
    Write (': ');
    ReadLn (Vorname);
    GoToXY (Anfangsspalte+Textbreite, Anfangszeile+4);
    Write (': ');
    ReadLn (Name);
    GoToXY (Anfangsspalte+Textbreite, Anfangszeile+5);
    Write (': ');
    ReadLn (Ort);
    ClrScr;
  END (* von Eingaben *);
```

Verdeckung äußerer Variablen durch prozedurinterne

Es ist übrigens kein Fehler, sondern durchaus zulässig und hier auch vernünftig, daß innerhalb der Prozedur dieselben Namen anders verwendet werden als außerhalb. Wenn zum Beispiel *Anfangszeile* in der Prozedur neu definiert wird, so „vergißt" das Programm einfach die Bedeutung, die der Name außerhalb hatte, solange die Prozedur arbeitet. Nach Verlassen der Prozedur wird die ursprüngliche Bedeutung wieder sichtbar.

Das Programmierschema E-V-A

Wenn zwischen dem Eingabeteil eines Programmes und dem Zeigen eines Ergebnisses (im Beispiel das fertige Schildchen) ein Berechnungsteil stünde, so hätten Sie ein klassisch aufgebautes Programm. Nach dem klassischen Schema durchläuft ein Programm nacheinander die Schritte Eingabe-Verarbeitung-Ausgabe (das sogenannte E-V-A-Schema). Wenn zum Beispiel die folgende Variante des Namensschildprogramms erstellt werden soll, bei welcher der umrahmte Name am rechten Rand des Schirms erscheint, so kann es zur Übersichtlichkeit beitragen, wenn von der Ausgabeprozedur *SchildMalen* die Berechnung der Lage und der Größe des Kastens abgetrennt wird (außerdem ist *GesamtlaengeUmrahmterText* jetzt als *Kastenlaenge* bezeichnet):

```
PROGRAM SchildAmRand;

USES crt, printer; {Muß erst ab den Versionen Turbo-Pascal 4.0
                    eingefügt werden.}
CONST
  Anfangszeile  =  8;
  Anfangsspalte = 20;
```

```
VAR
  Vorname, Name, Ort          : STRING [80];
  Kastenlaenge                : Integer;
  AbstandVomLinkenBildrand    : Integer;

 Eingaben;
  CONST
    Anfangszeile   =  2;
    Anfangsspalte  =  4;
    Textbreite     = 12;

 BEGIN
    ClrScr;
    GoToXY (Anfangsspalte, Anfangszeile);
    Write  ('Angaben für Namensschildchen');
    GoToXY (Anfangsspalte, Anfangszeile+1);
    Write  ('============================');
    GoToXY (Anfangsspalte, Anfangszeile+3);
    Write  ('Vorname');
    GoToXY (Anfangsspalte, Anfangszeile+4);
    Write  ('Nachname');
    GoToXY (Anfangsspalte, Anfangszeile+5);
    Write  ('Herkunftsort');
    GoToXY (Anfangsspalte+Textbreite, Anfangszeile+3);
    Write  (': ');
    ReadLn (Vorname);
    GoToXY (Anfangsspalte+Textbreite, Anfangszeile+4);
    Write  (': ');
    ReadLn (Name);
    GoToXY (Anfangsspalte+Textbreite, Anfangszeile+5);
    Write  (': ');
    ReadLn (Ort);
    ClrScr;
  END (* von Eingaben *);

PROCEDURE Berechnungen;
  BEGIN
    Kastenlaenge
    := Length (Vorname) + 1 + Length (Name) + 2 + Length (Ort);
    AbstandVomLinkenBildrand := 80 - (Kastenlaenge + 6);
                      {6 Zeichen zusätzlich sind von der
                       Bildschirmbreite (80 Zeichen) für den Rahmen
                       abzuziehen.}
  END (* von Berechnungen *);

PROCEDURE StrichMalen;
  VAR
```

```
    i: Integer;
  BEGIN
    FOR i := 1 TO Kastenlaenge DO
      Write ('-');
  END (* von StrichMalen *);

PROCEDURE SchildMalen;
  BEGIN (* SchildMalen *)
                          {Wie bisher}
  END (* von SchildMalen *);

BEGIN (* von SchildAmRand *)
  ClrScr;
  Eingaben;
  Berechnungen;
  SchildMalen;
END.
```

Ausgabe auf dem Drucker

Mit ein paar Änderungen kann man Sie sich die Namensschilder nicht nur per Hardcopy ausdrucken lassen, sondern direkt auf einen angeschlossenen Drucker schicken: Das ist bei diesem Programm nicht schwierig, weil die GoToXY-Anweisungen in der Ausgabe-Prozedur *SchildMalen* (anders als bei den Haus- und Stempelbeispielen) nur zum Einrücken dienen, aber nicht für einen Sprung nach oben verwendet werden.

Umlenken des Write zum Drucker

Das Wichtigste ist, daß die Write-Anweisung quasi auf den Drucker umgelenkt wird, wenn in der Klammer hinter Write als erstes eine entsprechende Angabe (gefolgt von einem Komma) steht.

Bis Turbo-Pascal 3.x genügt es, in jedes Write die Bezeichnung *LST* einzufügen. In der USES-Angabe hinter dem Programmkopf wird die Einheit *Printer* eingefügt, damit die Umleitung auch ab Turbo-Pascal 4.0 sowie in Quick-Pascal funktioniert. Ähnlich schiebt WriteLn auf dem Drucker das Papier eine Zeile weiter, statt den Cursor auf dem Schirm zum Anfang der folgenden Zeile zu dirigieren, wenn in Klammern das Ziel *LST* genannt wird. Die GoToXY-Anweisungen haben selber keine Wirkung auf den Drucker. Sie können entfernt werden. Die Papierbewegung wird allein durch WriteLn (LST) beziehungsweise Write (LST, {...}) gesteuert:

```
PROCEDURE StrichMalen;
  VAR
   i: Integer;
  BEGIN
    FOR i := 1 TO Kastenlaenge DO
      Write (LST, '-');
  END (* von StrichMalen *);

PROCEDURE SchildMalen;
  BEGIN (* von SchildMalen *)
    WriteLn (LST);
    WriteLn (LST);
    Write   (LST, '┌──');
    StrichMalen;
    Write   (LST, '──┐');
    WriteLn (LST);
    Write   (LST, '│  ');
    Write   (LST, Vorname, ' ', Name, ', ', Ort);
    Write   (LST, '  │');
    WriteLn (LST);
    Write   (LST, '└──');
    StrichMalen;
    Write   (LST, '──┘');
    WriteLn (LST);
  END (* von SchildMalen *);
```

Das Einrücken läßt sich mit einer Extra-Prozedur auch auf dem Drucker erreichen (die am besten als weitere Unterprozedur von *Schildmalen* eingefügt und für jede Zeile des Schildes aufgerufen wird):

```
PROCEDURE Einruecken;
  VAR
    i: Integer;
  BEGIN
    FOR i := 1 TO AbstandVomLinkenBildrand
    DO
      Write (LST, ' ');
  END (* von Einruecken *);
```

Man kann auch den Trick des formatierten Leerstrings verwenden (der auch auf dem Drucker funktioniert). Die folgende Programmzeile hat dieselbe Wirkung wie der Aufruf von *Einruecken*:

```
Write (LST, '': AbstandVomLinkenBildrand);
```

Im nächsten Kapitel erfahren Sie, wie in Pascal-Programmen Rechenausdrücke gebildet werden. Dann wird sich das Schildchen auch in die Mitte des Bildschirmes oder des Papiers positionieren lassen.

Aufgaben

1. Machen Sie Ihr Programm, das ein Haus zeichnet, zu einem Dialogprogramm. Sie können bei jedem Programmlauf bestimmen, wo die Tür sitzt, aus welchen Zeichen ein Fensterrahmen besteht, aus welchen Zeichen die

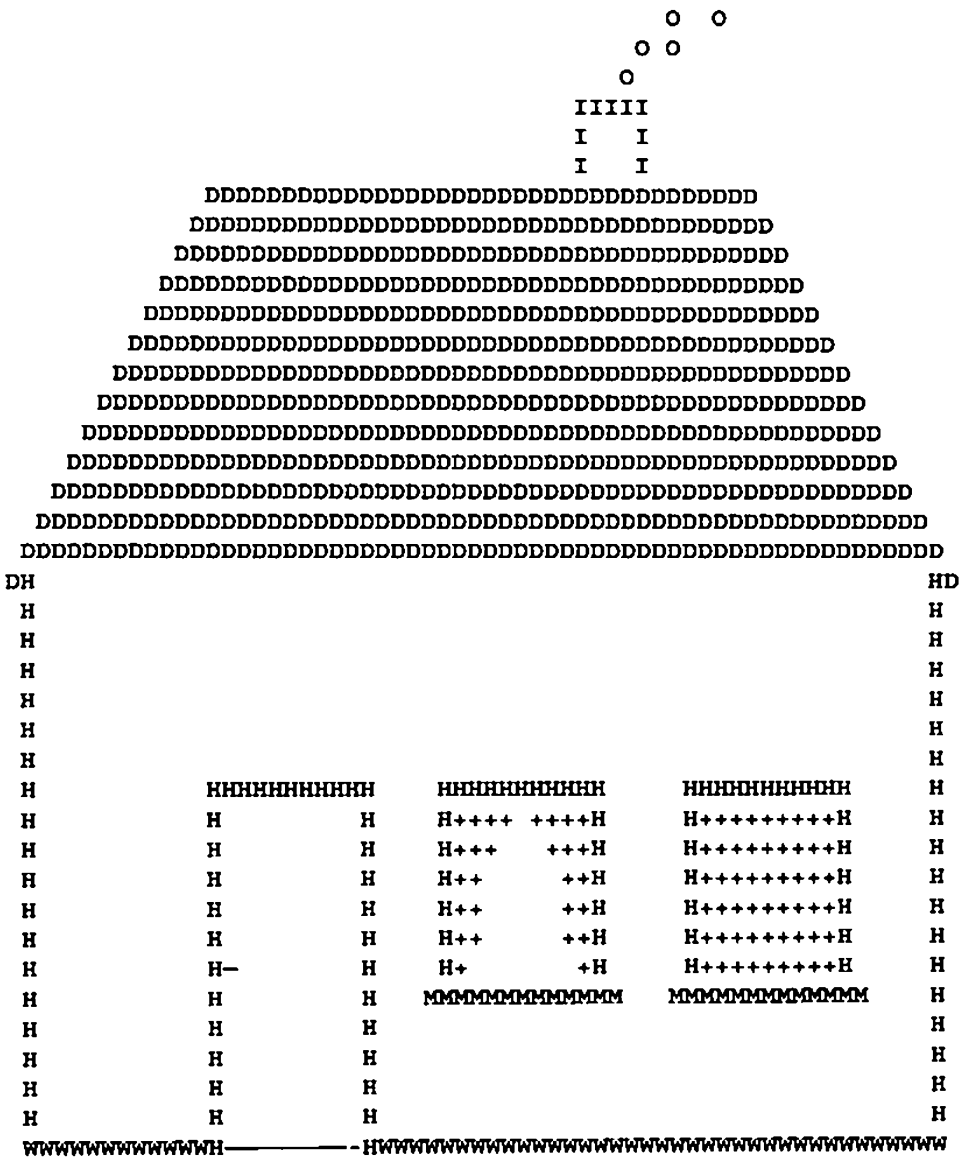

Abbildung 6-2: Ein Haus als „Dialogprogramm"

Fensterfüllung aufgebaut sein soll und ob die Vorhänge geöffnet oder geschlossen sind.

2. Erweitern Sie das Bahnhofprogramm um eine Eingabemöglichkeit für die Geschwindigkeit der Lokomotive (beziehungsweise für den Wert der in die Delay-Anweisung eingesetzten Verzögerungszeit).

3. Schreiben Sie ein Programm, das einen Lieferschein oder eine Rechnung auf dem Bildschirm (eventuell auch auf dem Drucker) gestaltet. Überlegen Sie, welche Daten Sie als Eingaben benötigen und was der Computer dann errechnen muß. Der gestaltete Vordruck kann zum Beispiel so aussehen:

```
                                        Buchversand Schmöker
                                        Postfach 0123
                                        1011 Unterdorf

                    Rechnung
                    --------

Wir lieferten:
2 Ex. Simmel, Maria: Mit Tränen und Klauen.

   Einzelpreis    DM 25.90
   Gesamtpreis                          DM 51.80
   Porto & Verp.                        DM  3.50
                                        --------
   Rechnungsbetrag                      DM 55.30
                                        ========

Zahlbar sofort auf Konto-Nr.: 6900,
Kreissparkasse Nebenacker, BLZ 500 300 01.

Gerichtsstand ist Nebenacker.
```

Abbildung 6-3: Rechnungsformular

7 Rechnen mit dem Computer

Addieren, Subtrahieren, Multiplizieren – Dividieren mit / – Die ganzzahlige Division mit DIV und MOD – Real und Integer – Zahlen und Zeichen – Ein neues Movie – Aufgaben

Zahlenwelten

Der Computer ist, wie Sie schon bemerkt haben, – trotz seines Namens – nicht einfach ein „Rechner". So wie die Mathematik nicht nur das Rechnen lehrt, sondern sich allgemein mit formalen, regelhaften Systemen (oder „Welten") beschäftigt, so dient der Computer dazu, (Zeichen-)Operationen in ganz unterschiedlichen Zusammenhängen und Formen auszuführen und dadurch jene eigene „Welten" zu schaffen.

Für viele Zwecke bildet dennoch in der Mathematik wie auf dem Computer das Rechnen mit Zahlen bzw. mit symbolischen Größen (das heißt, mit „Variablen") ein wichtiges Hilfsmittel. Wie dies in Pascal funktioniert, haben Sie schon gesehen: Von Anfang an haben Sie mit Zahlen und symbolischen Konstanten, später dann mit Variablen gearbeitet und den Computer das Ergebnis von Rechenausdrücken bestimmen lassen. Dabei lernten Sie bereits die Unterscheidung zwischen den Haupttypen kennen, die es in Pascal für Zahlen (-Variable) gibt: Integer für die ganzen Zahlen (genaugenommen nur für die zwischen –32768 und +32767) und Real für Dezimalzahlen. Im folgenden soll das bereits Gesagte ergänzt und systematisch dargestellt werden, wie Sie den Computer richtig rechnen lassen.

Addieren, Subtrahieren, Multiplizieren

Zahlen-Konstanten (zu denen auch die Zahlen selbst gehören) und Zahlen-Variablen können addiert, subtrahiert und auch multipliziert werden. Die Rechenoperationen werden genauso wie üblich geschrieben, nur für die Multiplikation dient „*" als „mal". Zum Beispiel können fol-

gende Rechenausdrücke oder „arithmetische Ausdrükke") gebildet werden:

```
Anfangszeile + 1
Endzeile - 3
2 * Hoehe - 1
```

„Punkt vor Strich"

Dabei gilt in Pascal wie in der Mathematik die Regel Punktrechnung geht vor Strichrechnung. Operationen mit Punktrechnungen (Multiplikation und Division) sind vor den Strichrechnungen Addition und Subtraktion auszuführen. Oben ist zum Beispiel zuerst der Wert von *Hoehe* zu verdoppeln und vom Ergebnis dieser Rechnung 1 abzuziehen.

Klammern

Es können auch eine praktisch unbegrenzte Anzahl von Klammern benutzt werden, um die Reihenfolge von Rechenoperationen zu bezeichnen. Zum Beispiel ist:

```
(2 * Hoehe) - 1
```

dasselbe wie der oben angeführte Ausdruck,
aber etwas anderes als:

```
2 * (Hoehe - 1)
```

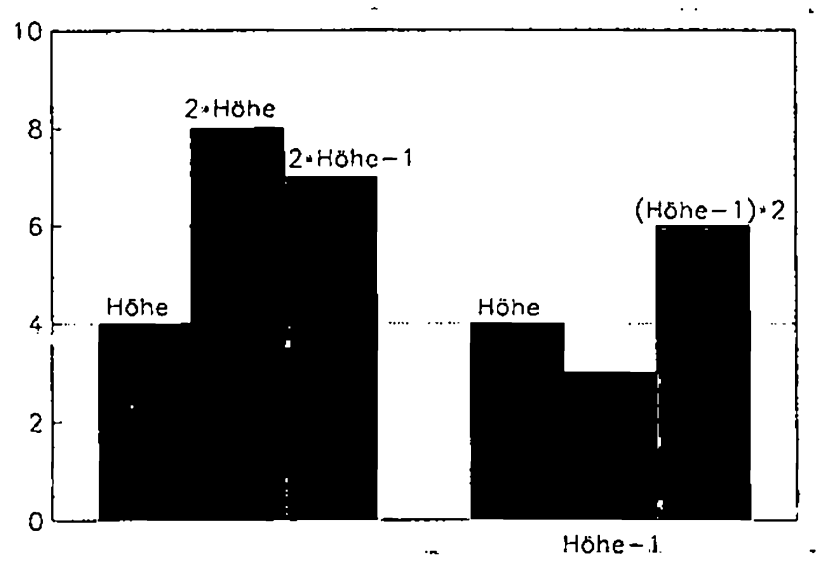

*Abbildung 7-1: Grafischer Vergleich von 2 * Hoehe – 1 und 2 * (Hoehe-1)*

Rechenprobleme auf dem Computer

Bei Addition und Multiplikation ist die Reihenfolge der beteiligten Größen theoretisch gleichgültig. Da der Computer nur mit begrenzter Stellenzahl arbeitet, können sich allerdings je nachdem, was zuerst gerechnet wird, unterschiedlich große Rundungsfehler einschleichen.

Die drei Operationen Addition, Subtraktion und Multiplikation haben gemeinsam, daß sie, auf Integer-Größen angewandt, auch wieder Integer-Größen ergeben, so wie in der Mathematik Summe, Differenz und Produkt von ganzen Zahlen wieder ganzzahlig sind. Gemischte Operationen von Integer- und Real-Größen sind zulässig. Aber:

Merkzettel

> Sobald in einem Rechenausdruck eine der beteiligten Größen den Typ Real hat, ist der ganze Ausdruck vom Typ Real. Eine Variable, an die das Ergebnis zugewiesen wird, muß entsprechend deklariert sein.

Ist zum Beispiel *Hoehe* eine Real-Variable und soll die Wertzuweisung:

```
Breite := 2 * Hoehe - 1
```

in einem Programm vorkommen, so muß auch Breite vom Typ Real sein. So etwas vergißt sich leicht. Der Compiler weist in Pascal dann mit der Fehlermeldung einer Typen-Unverträglichkeit darauf hin.

Dividieren mit /

Warum ist die Division etwas Schwierigeres als die anderen Rechenarten? In erster Linie liegt es daran, daß die Division aus dem Bereich der ganzen Zahlen herausführen kann. Schon 1 geteilt durch 2 ergibt 0.5, also keine ganze Zahl, obwohl 1 und 2 ganzzahlig sind. In Pascal wird diese Form der Division mit dem Schrägstrich „/" geschrieben (beispielsweise *Ergebnis := 1/2*). „1/2" bedeutet nicht „einhalb" gleich 0.5, sondern ist der noch nicht ausgerechnete Divisionsausdruck „1 geteilt durch 2". Es gilt:

Merkzettel

> Jeder Rechenausdruck, in dem eine Division mit / vorkommt, ist vom Typ Real.

Eine Variable wie *Ergebnis*, der ein solcher Ausdruck zugewiesen wird, muß auch dann als Real vereinbart sein, wenn rechts zufällig eine aufgehende Division wie 15/3 steht.

Auch bei der Division ist die Regel „Punktrechnung geht vor Strichrechnung" zu beachten. Nach der Zuweisung:

```
Ergebnis := 30 / 10 + 5
```

enthält *Ergebnis* den Wert 8 = 3 + 5 (und nicht etwa 2 = 30 / 15) und bei:

```
Ergebnis := 6 + 24 / 4
```

kommt 12 = 6 + 6 und keinesfalls 7.5 = 30/4 heraus.

Dabei bezieht sich der Schrägstrich (wie das Multiplikationszeichen) nur auf die direkt folgende Größe:

```
Ergebnis := 25 / 5 * 2
```

hat den Wert 10 = 25 · (1/5) · 2 und nicht etwa 2.5 = 25 / (5 · 2). Es muß also mit Klammern gearbeitet werden, wenn durch das Produkt oder die Summe von Größen geteilt werden soll:

```
Ergebnis := 12 / (3 + 5)
```

gibt 12/8 gleich 1.5, während

```
Ergebnis := 12 / 3 + 5
```

den Wert 4 + 5 gleich 9 hat.

Die ganzzahlige Division mit DIV und MOD

Sie werden aber überrascht feststellen, daß Sie für viele Fragestellungen die Division, die mit / geschrieben wird und auf Dezimalzahlen führt, gar nicht brauchen. Man arbeitet sehr oft mit etwas, das in der Schule schon vor den Dezimalzahlen gelehrt wurde und zu vielen Zwekken benötigt wird:

In eine Kassettenbox passen 15 Kassetten. Wieviele Boxen brauchen Sie, wenn Sie 83 Kassetten unterbringen wollen? Nicht 5.533, wie ein Taschenrechner sagt. Denn es gibt zwar 0.533 Liter oder 0.533 Meter, aber keine halben Boxen. Sondern Sie fragen, wie oft paßt die 15-Kassetten-Einheit ganz in die 83 hinein? Und was ist mit den übrigbleibenden Kassetten zu tun?

Division mit Rest

Die Rechenweise, die in der Grundschule als „Division mit Rest" den Schülern vermittelt wird, lautet etwa: 83

geteilt durch 15 ergibt 5 Rest 8. Sie benötigen also erst einmal fünf Boxen, acht einzelne Kassetten bleiben übrig. Was man mit diesem Rest anfängt, muß extra überlegt werden (man könnte sie unverpackt lassen, eine kleinere Schachtel suchen oder auch eine große halb leer lassen.)

DIV und MOD

Die Aufgabe hat eigentlich zwei Ergebnisse, die 5 und die 8. Um den ersten Wert herauszufinden (wie oft die eine Zahl ganz in die andere paßt), gibt es in Pascal die Operation DIV. Den Rest bei dieser ganzzahligen Division ermitteln Sie mit der Operation MOD: *83 DIV 15* hat den Wert 5; *83 MOD 15* ergibt dagegen 8. Diese Operationen sind nur für Integer-Größen möglich, der Ausdruck bleibt ganzzahlig. Mittels DIV und MOD können Sie in Pascal dividieren, ohne dabei den Integer-Zahlenbereich zu verlassen.

DIV und MOD gelten als Punktrechnungen wie Multiplikation und Division. Auch bezüglich der Bindung des nachfolgenden Divisors werden sie wie die Division behandelt.

Test: Aufgehen der Division

Eine besonders wichtige Anwendung von MOD ist die Frage, ob eine Division aufgeht. Wissen Sie zum Beispiel, welche Jahre (außer vollen Jahrhunderten) Schaltjahre sind, das heißt einen 29. Februar haben? Die Jahreszahl muß durch 4 teilbar sein. Mit MOD läßt sich das so ausdrücken, daß der folgende Ausdruck 0 ergeben muß:

```
Jahreszahl MOD 4
```

Oft können Sie auch den Rest der Division als Ausgangsbasis für eine neue Rechnung nehmen.

Beispiel

Betrachten Sie ein kleines Beispiel: Ein Programm soll für einen eingegebenen Betrag (vorläufig in vollen DM) ermitteln, in welchen möglichst großen Scheinen und Münzen der Betrag bar ausgezahlt werden kann. Was ist zu rechnen? Der Betrag dividiert durch 1000 ergibt die Zahl der benötigten Tausendmarkscheine, der Rest ist weiter zu bearbeiten, das heißt durch 500 zu teilen und so weiter:

```
PROGRAM Stueckelung;

USES crt, printer;

VAR
  Gesamtbetrag                    : Integer;
  Verbleibend                     : Integer;
```

```
  AnzahlTausender, AnzahlFuenfhunderter,
  AnzahlHunderter, AnzahlFuenfziger,
  AnzahlZwanziger, AnzahlZehner,
  AnzahlFuenfer,   AnzahlZweier,
  AnzahlEinMarkstuecke                    : Integer;

PROCEDURE ProgrammErklaeren;
  BEGIN
    ClrScr;
    WriteLn;
    WriteLn ('Das Programm ermittelt die Anzahl der Münzen');
    WriteLn ('und Geldscheine für einen vollen DM-Betrag.');
    WriteLn;
    Write ('----------------------------------------');
    Write ('----------------------------------------');
    WriteLn;
    WriteLn;
  END;

PROCEDURE Eingabe;
  BEGIN
    WriteLn ('Für welchen vollen DM-Betrag wollen Sie eine',
             'Stückelung ermitteln?');
    Write   ('Betrag: ');
    ReadLn  (Gesamtbetrag);
  END (*von Eingabe *);

PROCEDURE Berechnungen;
  BEGIN
    AnzahlTausender      := Gesamtbetrag DIV 1000;
    Verbleibend          := Gesamtbetrag MOD 1000;
    AnzahlFuenfhunderter := Verbleibend DIV 500;
    Verbleibend          := Verbleibend MOD 500;
    AnzahlHunderter      := Verbleibend DIV 100;
    Verbleibend          := Verbleibend MOD 100;
    AnzahlFuenfziger     := Verbleibend DIV 50;
    Verbleibend          := Verbleibend MOD 50;
    AnzahlZwanziger      := Verbleibend DIV 20;
    Verbleibend          := Verbleibend MOD 20;
    AnzahlZehner         := Verbleibend DIV 10;
    Verbleibend          := Verbleibend MOD 10;
    AnzahlZweier         := Verbleibend DIV 2;
    AnzahlEinMarkStuecke := Verbleibend MOD 2;
  END (*von Berechnungen*);

PROCEDURE Anzeige;
  BEGIN
```

```
    WriteLn;
    WriteLn ('Zahl 1000er: ', AnzahlTausender      :3);
    WriteLn ('Zahl  500er: ', AnzahlFuenfhunderter :3);
    WriteLn ('Zahl  100er: ', AnzahlHunderter      :3);
    WriteLn ('Zahl   50er: ', AnzahlFuenfziger     :3);
    WriteLn ('Zahl   20er: ', AnzahlZwanziger      :3);
    WriteLn ('Zahl   10er: ', AnzahlZehner         :3);
    WriteLn ('Zahl    2er: ', AnzahlZweier         :3);
    WriteLn ('Zahl    1er: ', AnzahlEinmarkstuecke :3);
    ReadLn;
  END (*von Anzeige*);

BEGIN (* Hauptprogramm *)
   ProgrammErklaeren;
   Eingabe;
   Berechnungen;
   Anzeige;
END.
```

Programmiermethodik

Fanden Sie dieses Problem schon etwas knifflig? Mit folgender Methode, an das E-V-A-Schema aus dem vorigen Kapitel anknüpfend, können Sie an manche Aufgaben systematisch herangehen: Überlegen Sie sich:

- Welche Größen sollen ermittelt werden (Ausgabegrößen)?
- Was ist dafür gegeben, beziehungsweise was brauchen Sie dafür (Eingabegrößen)?
- In welchen Rechenschritten (und mit welchen Zwischengrößen) kommt man von den Eingabe- zu den Ausgabegrößen?

Alle beteiligten Größen erscheinen dann im Programm als Variablen, auch die Rechenschritte können Sie ziemlich direkt ins Programm übernehmen.

Real und Integer

Das Programm *Stueckelung* wäre um einiges mühsamer zu schreiben, wenn es nicht die ganzzahligen Operationen DIV und MOD gäbe. Allerdings leidet die Ganzzahlrechnung immer unter der Beschränktheit des Zahlbereichs. Bei einem höheren Ausgangsbetrag als 32767 fängt das Programm zu „spinnen" an. Sie sehen, daß es erhebliche Auswirkungen haben kann, ob Sie sich entscheiden, Zahlen als Integer oder als Real anzusehen. In erster Linie

sollten Sie sich daran orientieren, was bei den jeweiligen gemessenen oder gezählten Gegenständen oder Größen sinnvoll ist: Eine Münze kann nicht, ohne ihre Gültigkeit zu verlieren, geviertelt werden, ein Zollstock (als Verkaufsartikel in einem Geschäft) auch nicht, wohl aber ein Wegkilometer (als Maß). Manchmal hängt es vom Zusammenhang oder sogar einer Rechtsverordnung ab, ob eine Größe als ganze Zahl oder als Dezimalzahl zu verstehen ist, zum Beispiel bei Zeugnisnoten. Überlegen Sie einmal bei den folgenden Beispielen, ob Sie sie als Integer- oder Real-Variable deklarieren würden:

```
Hosengroesse,          Hosenlaenge,
Durchschnittsnote,     Alter,
Menge_Schokolade       {in g},
Zeugnisnote,           Temperatur,
Anzahl_der_Haltestellen
```

Ausnahmen

Wenn allerdings die Zahlen größer werden als die maximal zulässige ganze Zahl für Integer-Werte, bleibt nichts übrig, als Zuflucht zu Real- Größen zu nehmen, die mehr Stellen zulassen.

Nicht tolerierbare Rundung

Die dabei auftretenden Ungenauigkeiten können allerdings oft nicht toleriert werden. Stellen Sie sich ein Buchhaltungsprogramm vor, das Beträge, seien es auch Pfennige, unterschlägt. Ein Buchhaltungsprogramm darf in keinem Fall versehentlich oder aus technischen Gründen, weil es intern Real-Zahlen verwendet, Beträge verschlukken – und seien sie noch so klein.

Spezialkonstruktionen zur Rechnung mit großen Ganzzahlen

Um mit großen Zahlen, bei denen nicht gerundet werden darf, arbeiten zu können, haben sich die Konstrukteure der meisten Pascal-Systeme Spezialitäten ausgedacht. In vielen Pascal-Arten gibt es einen eigenen Datentyp für lange Ganzzahlen. Ein solcher Typ ist ab Turbo 4.0 unter dem Namen *LongInt* vordefiniert. Anstelle der 16 Bit der gewöhnlichen Integer-Variablen wird für Werte dieses Typs der doppelte Speicherbereich reserviert. LongInt-Werte dürfen daher im Bereich von –2147483648 bis +2147483647 liegen, also zirka zwischen ± 2 Milliarden. Damit läßt sich die Buchhaltung eines kleineren Betriebes durchaus programmieren. Wenn Sie über Turbo-Pascal ab Version 4.0 und einen numerischen Koprozessor oder über Turbo-Pascal ab Version 5.0 bzw. Quick-Pascal verfügen, so gibt es für derartige Aufgaben einen noch besser geeigneten Spezialtyp, den Typ Comp. Bei Comp wird praktisch nochmals doppelt so viel Spei-

cher wie bei LongInt für eine ganze Zahl reserviert, so daß Zahlen zwischen -2^{63} und $+2^{63}-1$ (und damit noch 18stellige ganze Zahlen) dargestellt werden können. Voraussetzung ist, daß der Koprozessor (bzw. ab Turbo 5.0, falls nötig, stattdessen seine Software-Emulation) am Anfang des Programms mittels der Compileroption {$N+} aufgerufen wird.

Hin und zurück zwischen den Typen

Die Bereiche Real und Integer sind nicht völlig gegeneinander abgeschottet: Jeder Ausdruck von Integer-Größen, in denen eine Division mit / vorkommt, wird bekanntlich zu einem Real-Ausdruck. Außerdem darf jeder Integer-Ausdruck an eine Real-Variable zugewiesen werden. Um eventuell den umgekehrten Weg beschreiten zu können, gibt es zwei Funktionen:

```
trunc (Zahl)
```

schneidet von der Real-Größe *Zahl* die Stellen hinter dem Komma ab (truncate, „stutzen", eigentlich „auf den Baumstumpf zurückschneiden"). Das Ergebnis ist der entsprechende ganzzahlige Integer-Wert. Zum Beispiel bedeutet:

```
trunc (23.2)
trunc (-4.5)
```

die Zahl 23 bzw. die ganze Zahl –4. Die zweite Funktion zum Runden heißt:

```
round (Zahl)
```

und ergibt den gerundeten Wert von *Zahl* als Integer-Größe.

```
round (23.2)
round (-4.5)
```

ist 23 bzw. -5.

Es soll jetzt mit Hilfe der *trunc*-Funktion das Programm *Stueckelung* erweitert werden, so daß es Beträge größer als +32767 zunächst als Real-Zahlen verarbeitet. Die Kniffe, die dabei angewendet werden, gehören schon zu den schwierigeren Tricks. Verzweifeln Sie nicht, wenn Ihnen vorläufig so etwas nicht selbst einfällt.

Verbessertes Stückelungsprogramm

Hier ist erst einmal das neue Programm, so weit es vom alten abweicht:

```
PROGRAM Stueckelung;
USES crt, printer;

VAR
  Gesamtbetrag, VerbleibenderBetrag : Real;
  Verbleibend : Integer;
  AnzahlTausender, AnzahlFuenfhunderter,
  AnzahlHunderter, AnzahlFuenfziger,
  AnzahlZehner, AnzahlZwanziger,
  AnzahlFuenfer, AnzahlZweier,
  AnzahlEinMarkstuecke : Integer;

PROCEDURE ProgrammErklaeren;
  BEGIN
                          {Wie bisher}
  END;

PROCEDURE Eingabe;
  BEGIN
                          {Wie bisher}
  END (*von Eingabe *);

PROCEDURE Berechnungen;
  BEGIN
    AnzahlTausender     := trunc (Gesamtbetrag/1000);
    VerbleibenderBetrag := Gesamtbetrag - AnzahlTausender * 1000.0;
    Verbleibend         := trunc (VerbleibenderBetrag);
                           {Die trunc-Funktion überführt hier den
                           Real-Wert in einen Integer-Wert;
                           ab dann geht es weiter im
                           Integer-Bereich.}
    AnzahlFuenfhunderter := Verbleibend DIV 500;
    Verbleibend          := Verbleibend MOD 500;
    AnzahlHunderter      := Verbleibend DIV 100;
    Verbleibend          := Verbleibend MOD 100;
    AnzahlFuenfziger     := Verbleibend DIV 50;
    Verbleibend          := Verbleibend MOD 50;
    AnzahlZwanziger      := Verbleibend DIV 20;
    Verbleibend          := Verbleibend MOD 20;
    AnzahlZehner         := Verbleibend DIV 10;
    Verbleibend          := Verbleibend MOD 10;
    AnzahlZweier         := Verbleibend DIV 2;
    AnzahlEinMarkStuecke := Verbleibend MOD 2;
  END (* von Berechnungen *);
                         {Weiter wie bisher}
```

In den Berechnungen wird zunächst die Anzahl der vollen Tausender ermittelt, indem *Gesamtbetrag* mittels der für Real-Werte allein möglichen Operation / durch 1000 geteilt wird. Vom Ergebnis wird durch Abschneiden der Stellen hinter dem Komma ein Integer-Wert gebildet. Der verbleibende Betrag, der weiter aufzuschlüsseln ist, ergibt sich dadurch, daß – als Ersatz für eine im Real-Bereich nicht vorhandene MOD-Operation – die Differenz des Ausgangsbetrages und des durch Tausender beglichenen Betrages gebildet wird. Es muß hier:

```
AnzahlTausender * 1000.0
```

und nicht einfach 1000 geschrieben werden. Denn 1000 ist ganzzahlig, so daß das Produkt von *AnzahlTausender * 1000* auch Integer wäre, also nie +32767 überschreiten könnte. *1000.0* gilt dagegen als Real, so daß der ganze Ausdruck im Real-Bereich angesiedelt wird. Um anschließend wieder Integer-Operationen zu verwenden, wird noch *VerbleibenderBetrag* mittels trunc in eine Integer-Größe umgewandelt. Wegen der strengen Typenunterscheidung in Pascal muß so etwas manchmal sein: Eine Größe wird in eine andere überführt, wobei sich nur der Typ, aber nicht der Wert ändert.

Pascal kennt noch zwei weitere Standard-Transferfunktionen. Sie vermitteln zwischen dem Typ Char und dem Typ Integer.

Zahlen und Zeichen

Beim Malen der Rahmen zum Programm *Stempel* wurde deutlich, daß es Zeichen gibt, die nicht (beziehungsweise nur als Controlzeichen) im Editor geschrieben, aber mittels ihrer Nummer im ASCII-Zeichensatz angesprochen werden können.

chr-Funktion

Wenn die Nummer eines Zeichens durch eine Integer-Variable oder einen entsprechenden Ausdruck angegeben wird, benötigt man die Funktion *chr* („chr" soll an Char erinnern). Zum Beispiel bedeutet im IBM-Zeichensatz das Herz ♥:

```
chr (3)
```

Mit der folgenden Schleife können Sie sich den gesamten Zeichensatz ihres Computers vorführen lassen (wundern Sie sich nicht über ein paar merkwürdige Effekte bei

den ersten 32 Zeichen, den sogenannten Steuerzeichen Ihres Computers):

```
FOR Nummer := 0 TO 255 DO
  BEGIN
    WriteLn ( Nummer :3, ' ',  chr (Nummer)):
  END;
```

chr überführt also Zahlen in Char-Werte.

ord-Funktion

Die Umkehrung geht mit der Funktion *ord*, welche die Ordnungszahl, das heißt die Nummer eines Zeichens in der ASCII-Tabelle angibt. Zum Beispiel hat *ord ('A')* den Wert 65, *ord ('1')* den Wert 49. Beachten Sie den Unterschied zwischen dem Zahlzeichen '1' (Char-Wert) und der Zahl 1! Mit folgendem kleinem Programm können Sie irgendeine Taste oder Tastenkombination auf der Tastatur drücken und erfahren, welcher Codenummer auf der ASCII-Tabelle ihr entspricht. Probieren Sie auch die Control-Kombinationen (Tastenkombinationen mit der Strg-Taste) aus.

```
PROGRAM TastaturErforschen;

USES crt, printer;

VAR
  GedruecktesZeichen : Char;
  Codenummer         : Integer;

PROCEDURE ProgrammErklaeren;
  BEGIN
    ClrScr;
    WriteLn;
    WriteLn ('Das Programm ermittelt den Scancode einer Taste.');
    WriteLn ('Besondere Codes werden nicht dargestellt',
             '(Codenummer: 0).');
    WriteLn ('Bitte drücken Sie eine Taste Ihrer Wahl.');
    WriteLn;
    Write ('----------------------------------------');
    Write ('----------------------------------------');
    WriteLn;
    WriteLn;
  END;

BEGIN
  ProgrammErklaeren;
  Write ('Taste: ');
  GedruecktesZeichen := ReadKey;
```

```
                    {Diese Zeile muß in Turbo 3.x folgendermaßen
                    lauten: Read (KBD, GedruecktesZeichen);}
  Write (GedruecktesZeichen, '  ');
  Codenummer := ord (GedruecktesZeichen);
  Write ('Codenummer: ');
  Write (Codenummer:3);
  WriteLn;
END.
```

Direkte Eingaben von Einzelzeichen

Sie haben hier zum ersten Mal eine spezielle Tastaturlesefunktion *ReadKey* („Lies Taste") in das Programm eingebaut (ab Turbo-Pascal 4.0 vorhanden). ReadKey liest bei jedem Aufruf ein Zeichen von der Tastatur, ohne es automatisch auf dem Bildschirm auszugeben.

Für Spezialtasten liefert der erste Aufruf von ReadKey das Zeichen #0 zurück, der zweite Aufruf den sogenannten Scancode der Taste. Mittels des Scancodes, praktisch der Tastennummer, läßt sich dann identifizieren, welche Spezialtaste betätigt wurde. Zum Beispiel entspricht die Funktionstaste F1 dem Zeichen #59, F2 dem Zeichen #60 usw. Mehr hierzu in Kapitel 11, Seite 182.

Von STRINGs zu Zahlen

Mit Hilfe von ord ist es auch möglich, eine als STRING vorliegende Zahl in einen Integer-Wert zu verwandeln. Ist *Zahltext* eine STRING-Variable, deren Inhalt eine ganze Zahl darstellt, und *Zahl* eine Integer-Größe, so kommt mit den folgenden Anweisungen der Wert der Zahl auf *Zahl*. Als Hilfsgrößen finden die Integer-Variablen *i*, *LetzteStelle* und *Ziffernwert* Verwendung.

```
LetzteStelle := Length (Zahltext);
Zahl         := 0;
FOR i := 1 TO LetzteStelle DO
  BEGIN
    Ziffernwert := ord (Zahltext [i]) - 48;
    Zahl := Ziffernwert + 10 * Zahl;
  END;
```

Spielen Sie anhand eines Beispiels durch, wie diese Anweisungen arbeiten. Für die erste bis zur letzten Position des STRING-Wertes wird jeweils der Ziffernwert berechnet, indem mit der ord-Funktion vom Ziffernzeichen auf die dazugehörige Nummer übergegangen wird. Vergewissern Sie sich durch einen Blick auf die ASCII-Tabelle, daß die Nummer eines Ziffernzeichens tatsächlich um 48

höher ist, verglichen mit seinem Wert als Zahl. Der Ziffernwert der jeweiligen Stelle wird dann zur vorher schon errechneten Zahl hinzugefügt, nachdem diese um eine Zehnerstelle größer gemacht wurde. Zum Beispiel vom STRING '355' wird erst die 3 genommen und auf *Zahl* gebracht. Im zweiten Schritt addiert sich die erste 5 zu 30 = 10 * 3, schließlich die zweite 5 zu 350 = 10 * 35, so daß tatsächlich als *Zahl* am Schluß 355 herauskommt. Es wurde also die Umwandlung eines Strings in eine Zahl programmiert. In Turbo-Pascal gibt es dafür übrigens auch eine (bei der Version 3.x mit Vorsicht zu genießende) fertige Prozedur, die Sie später zusammen mit einer Prozedur für den umgekehrten Vorgang kennenlernen werden.

Ein neues Movie

Zum Schluß soll anhand einer Variante des Bahnhofsbeispiels gezeigt werden, wie mit geringem Aufwand durch geschickten Einsatz von etwas Formeln schon interessante Effekte produziert werden können: Im Programm Zugverkehr herrscht auf den beiden auf dem Bildschirm abgebildeten Gleisen ein reger Verkehr in zwei Richtungen.

```
PROGRAM Zugverkehr;

USES crt, printer;


CONST
  HoeheGleis1       =    8;
  HoeheGleis2       =   12;
  Gesamtzeit        = 5000;
  Geschwindigkeit1  =   10;
  Geschwindigkeit2  =   19;

VAR
  Streckenpunkt1    : Integer;
  Streckenpunkt2    : Integer;

PROCEDURE ProgrammErklaeren;
  BEGIN
    ClrScr;
    WriteLn;
    WriteLn ('Dieses Programm simuliert mit einer einfachen',
             'Grafik:');
    WriteLn;
    WriteLn ('       »Zugverkehr auf zwei Bahngleisen« ');
```

```
    WriteLn;
    WriteLn ('      Bitte drücken Sie die Taste [RETURN].');
    Write ('----------------------------------------');
    Write ('----------------------------------------');
    WriteLn;
    WriteLn;
  END;

PROCEDURE GleiseZeichnen;
  VAR    i: Integer;
  BEGIN
    GoToXY (1, HoeheGleis1);
    FOR i := 1 TO 80 DO
      Write ('_');
    GoToXY (1, HoeheGleis2);
    FOR i := 1 TO 80 DO
      Write ('_');
  END (*von GleiseZeichnen*);

PROCEDURE Fahren;
  VAR Zeit: Integer;
  BEGIN
    FOR Zeit := 1 TO Gesamtzeit DO
      BEGIN
        Streckenpunkt1 := (Zeit DIV Geschwindigkeit1) MOD 78 + 1;
        GoToXY (Streckenpunkt1, HoeheGleis1);
        Write ('_■');
        GoToXY (78, HoeheGleis1);
        Write ('__');
        Streckenpunkt2 := (Zeit DIV Geschwindigkeit2) MOD 78 + 1;
        GoToXY ( 80 - Streckenpunkt2, HoeheGleis2);
        Write ('■_');
        GoToXY (1, HoeheGleis2);
        Write ('__');
      END;
  END (*von Fahren*);

BEGIN
  ProgrammErklaeren;
  ReadLn;
  ClrScr;
  GleiseZeichnen;
  Fahren;
END.
```

Im nächsten Kapitel sehen Sie, wie man den Zug über Weichen leiten kann. Je nach dem Wert einer Variablen darf dann etwas Unterschiedliches passieren.

Abbildung 7-2: „Zugverkehr"

Aufgaben

1. Was kommt heraus bei:

```
14 MOD 3          17 DIV 3
41 MOD 10         41 DIV 10
45 DIV 9          45 MOD 9
121 MOD 11        1984 MOD 4
```

2. Wandeln Sie eine Zeitangabe in Sekunden in eine solche mit ganzen Stunden, Minuten und Sekunden um. Beispiel: Wieviel Stunden, Minuten, Sekunden sind 5111 Sekunden?

3. Schreiben Sie ein Programm, in dem folgende Ausdrücke einer Real-Variablen *Ergebnis* zugewiesen werden und anschließend *Ergebnis* mit zwanzig Nachkommastellen ausgegeben wird. Rechnen Sie nach: Bei allen Ausdrücken müßte das Ergebnis eigentlich 0 lauten!

a) ((1/5) * 5) – 1

b) 1.34 + 1.66 – 3.0

c) (((1234.99 – 1235.0) * 10000.0) + 100.0) * 100000.0

Turbo-Pascal bis zur Version 3.0 zeigt bereits bei den sehr simplen Ausdrücken a) und b) erstaunliche Fehler, die letztlich auf Ungenauigkeiten der Umwandlung der Dezimalzahlen in die interne Darstellung beruhen. Der dritte Ausdruck wird auch von den neueren Pascal-Compilern falsch berechnet, da er an die Grenze der Auflösung des Typs Real führt. Noch bei Turbo 4.0 ist der Fehler unannehmbar hoch – es kommt eine Zahl nahe bei –1.6 heraus! Die neueren Versionen werden immer besser – schauen Sie sich an, was Ihr Compiler liefert. Probieren Sie, wenn Sie über Turbo-Pascal ab 5.0 oder über Quick-Pascal verfügen, auch aus, was passiert, wenn Sie für *Ergebnis* statt des Typs Real die Typen Double oder Extended benutzen (dazu muß, wie oben beim Typ Comp erwähnt, der numerische Koprozessor angesprochen werden, indem am Programmanfang die Compileroption {$N+} eingefügt wird.)

4. Versuchen Sie das Programm, das einen Geldbetrag entsprechend den existierenden Scheinen und Münzen stückelt, zu erweitern, so daß auch Beträge mit Pfennigen verarbeitet werden.

5. Ergänzen beziehungsweise verändern Sie ein Lieferschein-Programm so, daß die Mehrwertsteuer vom Programm berechnet und ausgewiesen wird. Denken Sie dabei an die Formatierung der Ausgabe von Real-Größen, sonst wird die bei DM-Beträgen völlig unsinnige Gleitpunktdarstellung angezeigt.

6. Schreiben Sie ein Programm, das die Prozentanteile der Parteien bei den Wahlen als Balkendiagramm veranschaulicht. Einzugeben sind die erreichten beziehungsweise hochgerechneten Prozentsätze für CDU, SPD, Grüne, FDP, Sonstige.

```
|  CCCCCCC    SSSSSSS
|  CCCCCCC    SSSSSSS
|  CCCCCCC    SSSSSSS
|  CCCCCCC    SSSSSSS
|  CCCCCCC    SSSSSSS
|  CCCCCCC    SSSSSSS
|  CCCCCCC    SSSSSSS
|  CCCCCCC    SSSSSSS
|  CCCCCCC    SSSSSSS
|  CCCCCCC    SSSSSSS
|  CCCCCCC    SSSSSSS
|  CCCCCCC    SSSSSSS
|  CCCCCCC    SSSSSSS
|  CCCCCCC    SSSSSSS    GGGGGGG
|  CCCCCCC    SSSSSSS    GGGGGGG
|  CCCCCCC    SSSSSSS    GGGGGGG    FFFFFFF
|  CCCCCCC    SSSSSSS    GGGGGGG    FFFFFFF    XXXXXXX
|_____________________________________________________
   CDU         SPD        Grüne      FDP       Sonst.
```

Abbildung 7-3: Balkendiagramm in Zeichensatzgrafik

8 Die Entscheidungsstruktur

Sequenz und Entscheidungsstruktur – Die IF-THEN-ELSE-Anweisung – Bedingungsausdrücke – Syntax der IF-Anweisung – Programmablaufpläne und Struktogramme – Einseitige und geschachtelte Strukturen – Aufgaben

Rinks und Lechts kann man leicht velwechsern.
Ernst Jandl

Wer die Bedingungen stellt, bestimmt, wo es lang geht. Aber einen Weg zurück gibt es nicht. In der Theorie des Programmierens wird diese Programmfigur Entscheidungsstruktur oder auch Auswahlstruktur genannt. Die folgende Darstellung soll den Gedanken veranschaulichen: Der Weg, der Programmfluß, gabelt sich, eine Abzweigung zurück wird nicht zugelassen.

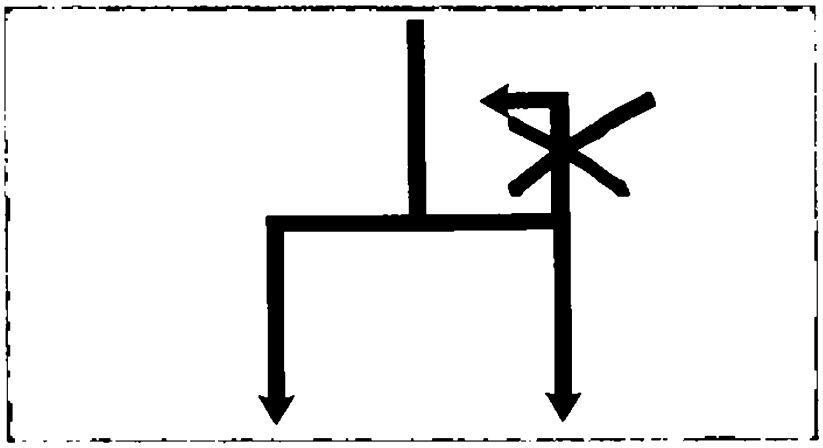

Abbildung 8-1: Der Programmfluß gabelt sich.

Sequenz und Entscheidungsstruktur

Die bisherigen Programme waren, abgesehen von der etwas aus dem Schema tanzenden FOR-Anweisung, so beschaffen, daß sie linear abgearbeitet wurden. Eine Anweisung folgte nach der anderen, wenn alle durch waren, endete das Programm. Diese einfache Programmstruktur trägt in der Theorie auch den Namen sequentielle Programmstruktur (oder auch Folge, lineare Folge oder Sequenz). Nach DIN läßt sich ein Diagramm für eine Sequenz aus zum Beispiel drei Anweisungen so zeichnen:

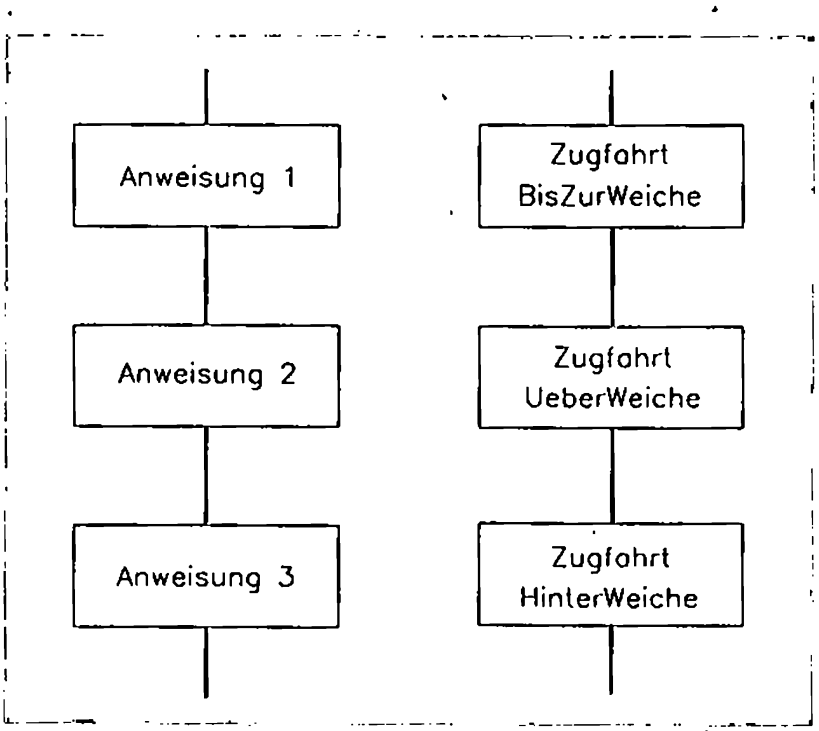

Abbildung 8-2: Sequentielle Programmstruktur: Schema und Beispiel „Zugfahrt"

Wollen Sie von dem Geradeaus der Sequenz abbiegen, je nach Umständen den einen oder den anderen Weg nehmen, Weichenstellungen in das Programm einbauen, so sagt die Theorie dazu: eine Entscheidungsstruktur verwenden.

Bedingungen ...

Dazu wird eine Bedingung formuliert (zum Beispiel: „Steht die Weiche auf Linksfahrt"). Ist sie erfüllt, soll der eine Weg (das heißt die eine Anweisung oder eine Folge von Anweisungen) eingeschlagen werden, andernfalls der zweite.

... und Konsequenzen

Ein anderes Beispiel wäre: Wenn eine Person Alkohol getrunken hat, nimmt sie (das legen Sie jetzt per Programm fest) ein Taxi, andernfalls fährt sie mit dem eigenen Wagen. Sie müssen sich aber vorsehen: Sie sollten sich beim Programmieren nicht nur über bestimmte eingetretene Umstände Gedanken machen, sondern über alle zulässigen Ereignisse. Es gibt Züge, die links abbiegen, und solche, die geradeaus weiterfahren. Da Programme sich eigentlich nur mit überschaubaren Situationen beschäftigen dürften, fordert die Entscheidungsstruktur von Ihnen, daß sich die aufgeteilten Zug-, Lebens- beziehungsweise Programmpfade stets wieder vereinigen. Alle Situationen münden (mindestens formell) wieder in das gleiche, gemeinsame Programm – andernfalls gibt es einen Programmabsturz. Die symbolische Darstellung

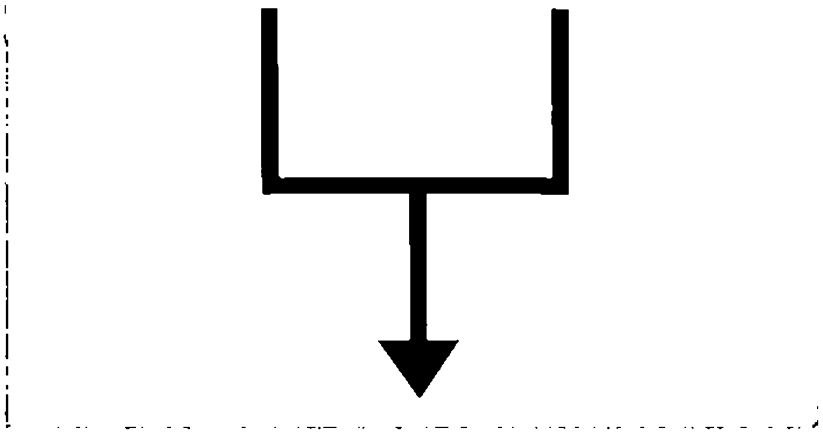

Abbildung 8-3: Der Programmfluß wird wieder zusammengeführt.

von oben bedarf daher noch der Fortsetzung. Die beiden Zweige müssen wieder zusammenkommen (Abb. 8-3).

Ein DIN-Diagramm für die Entscheidungsstruktur sieht (zum Beispiel mit den Anweisungen 1A, 1B wie „Taxi rufen", „Taxi nehmen" für den Pfad 1 beziehungsweise 2A, „mit eigenem Wagen fahren", für den Pfad 2) so aus:

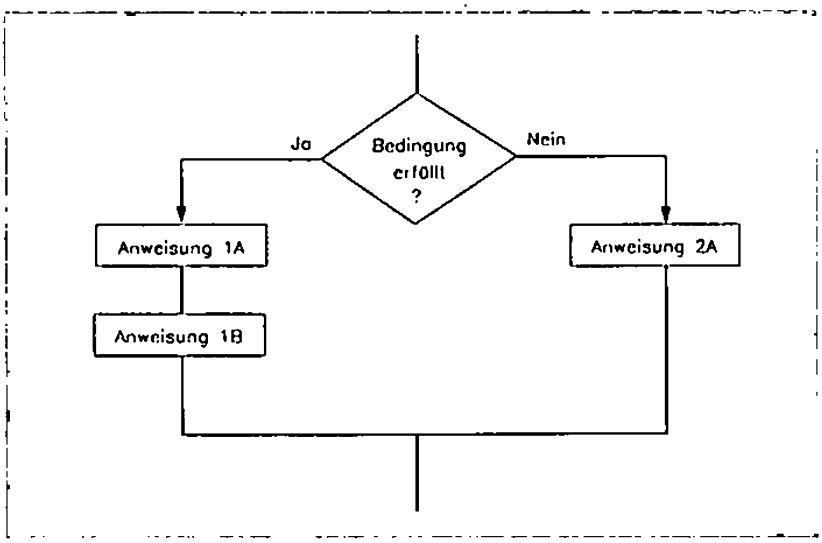

Abbildung 8-4: Diagramm nach DIN für eine Entscheidungsstruktur

Die IF-THEN-ELSE-Anweisung

In der Programmiersprache Pascal bildet man die Entscheidungsstruktur mit Hilfe der Schlüsselwörter IF („wenn"), THEN („dann") und ELSE („sonst, andernfalls"). Der Dann-Teil entspricht dem JA-Zweig, der Sonst-Teil dem NEIN-Zweig des Diagramms.

```
IF                          {Wenn ... Bedingungsausdruck}
  THEN                      {dann}
    BEGIN
      Anweisung1A;          {eine oder mehrere Anweisungen}
      Anweisung1B;
    END
  ELSE                      {sonst}
    BEGIN
      Anweisung2A;          {eine/mehrere Anweisungen}
    END;
```

Das ganze Schema stellt insgesamt eine IF-THEN-ELSE- oder kurz eine IF-Anweisung dar. Die Anweisungsgruppen des Dann- und des Sonst-Teils sind jeweils von BEGIN und END umschlossen. Mit Einrücken verdeutlicht man, welche Anweisungen zum DANN und zum SONST gehören, der Compiler erkennt es (wie schon bei der FOR-Anweisung) durch die Umrahmungen mit BEGIN-END. Die Anweisungen, die nach der gesamten IF-Anweisung (das heißt nach dem Abschluß von deren ELSE-Teil) folgen, werden unabhängig vom Zutreffen oder Nichtzutreffen der Bedingung ausgeführt. Im Sinne des Liniendiagramms sind ab hier die beiden Zweige wieder zusammengeführt. Ein Beispiel könnte etwa so aussehen:

```
IF Promille > 0
  THEN
    BEGIN
      TaxiRufen;
      TaxiNehmen;
    END
  ELSE
    BEGIN
      MitEigenemWagenFahren;
    END;
ZuBettGehen;
```

Promille ist dabei natürlich eine Real-Variable; *TaxiRufen*, *TaxiNehmen*, *MitEigenemWagenFahren* und *ZuBettGehen* sind Prozeduren.

Oder betrachten Sie ein anderes, unmittelbar verständliches Beispiel für einen kurzen Programmausschnitt mit einer IF-Anweisung (wobei *AnzahlKopien* eine Integer-Variable und *Kopienpreis*, *Kopienkosten* Real-Größen sind):

```
IF AnzahlKopien < 100
  THEN
    BEGIN
      Kopienpreis := 0.10;
    END
  ELSE
    BEGIN
      Kopienpreis := 0.09;
    END;                    {Hier ist die IF-Anweisung zu Ende.}
Kopienkosten := AnzahlKopien * Kopienpreis;
                            {Das trifft wieder auf beide Fälle zu.}
```

Bedingungsausdrücke

In den Beispielen wurde als Bedingungen angenommen und eingesetzt:

Promille > 0.8 beziehungsweise *AnzahlKopien < 100*. Bei einer IF-Anweisung darf als Bedingungsausdruck alles stehen, was der Computer auf zutreffend oder nichtzutreffend (wahr oder falsch) prüfen kann.

Vergleichs operatoren

In erster Linie sind dies Vergleiche, die mit Vergleichsoperatoren, das heißt > (größer), < (kleiner), >= (größer oder gleich), <= (kleiner oder gleich), <> (ungleich) oder einfach mit = gebildet werden.

Merkzettel

Vorsicht: Verwechseln Sie nicht die Wertzuweisung an eine Variable, die in Pascal mit := geschrieben wird, mit dem Vergleich dieser Variable mit einem Wert.

Vergleich nur bei kompatiblen Ausdrücken

Sie müssen bei der Bildung der Ausdrücke (ähnlich wie bei Wertzuweisungen) darauf achten, daß Sie, wie schon der Volksmund fordert, nicht Äpfel mit Birnen vergleichen. Nur für Größen desselben beziehungsweise eines kompatiblen Typs gibt es Vergleichsmöglichkeiten, also Real-Variablen mit Real-Zahlen, Char-Variablen mit Char-Werten und so weiter. Mit entsprechend deklarierten Variablen oder Konstanten dürfen Sie zum Beispiel schreiben:

```
Geschwindigkeit <  100
Gewicht         >= 80
Weichenstellung =  Abbiegestellung
Zeichen         =  'A'
Gewicht1        =  Gewicht2
```

```
Promille        >  0.8
Tempo           >  Limit + Limit/10
Promille        >  PromilleAbsoluteFahruntuechtigkeit
```

Die Richtung, in der Sie einen Ausdruck notieren, ist grundsätzlich gleichgültig. Sie könnten also genausogut *100 > Geschwindigkeit* und so weiter schreiben (anders als bei einer Wertzuweisung).

Boolean

Bei jedem dieser Ausdrücke kann der Computer feststellen, ob er „wahr" oder „falsch" ist (mit Sinn natürlich nur, wenn auf die Variablen vorher die richtigen Werte gebracht wurden). Deshalb heißen solche Ausdrücke auch logische oder boolesche Ausdrücke. Es gibt in Pascal einen eigenen Variablentyp für boolesche Werte, den Typ Boolean. Wie Sie im nächsten Kapitel sehen werden, kann eine boolesche Variable das Ergebnis eines Vergleichs (das heißt true oder false) aufnehmen und auch für sich einen Bedingungsausdruck bilden.

Logische Verknüpfungen

Bedingungen können mit logischen Verknüpfungen wie AND („Und", hier frei: „sowohl das eine wie das andere"), OR (nicht ausschließendes „Oder") und NOT („Nicht") zu neuen Bedingungen kombiniert werden. Beachten Sie: Die Verneinung NOT bezieht sich auf einen kompletten Ausdruck (nicht bloß wie in der Alltagssprache auf den Vergleichsoperator). Teilausdrücke sind in runde Klammern zu setzen. Drei Beispiele verdeutlichen das Gesagte:

```
(Alter >= 18) AND (Geschlecht = 'm')
                          {Gezogenwerden vom Militär möglich}
(Tempo > Limit) OR (Sichtweite < Bremsweg)
                          {Auto fährt zu schnell}
NOT (Taste = 'j')
                          {Andere Taste als 'j' gedrückt}
```

Alltagssprache und Logiksprache

Seien Sie vorsichtig, wenn Sie *Und, Oder* im Sinn der Verknüpfungen der Logik verwenden, welche sich von den Verknüpfungen des Alltagsgebrauchs unterscheiden. So sagt man: „Ich kann mit der Taxe und (auch) mit dem Bus zum Bahnhof fahren." Man kann in dieser Weise mitteilen, daß beide Alternativen möglich sind (eventuell die eine gegenüber der anderen bevorzugt wird). Daß nicht beides gleichzeitig zu passieren braucht, ist selbstverständlich, weil man weiß, daß es nicht gleichzeitig geschehen kann. Formallogisch müssen Sie jedoch genau diese Information ausdrücken, daß man mit dem einen

oder dem anderen Verkehrsmittel fährt. Im Pascal-Stil ist zu notieren:

```
(Verkehrsmittel = Taxi) OR (Verkehrsmittel = Bus)
```

Angabe eines Char-Bereichs

Für Char-Größen gibt es eine sehr praktische Schreibweise, wenn Sie einen Bereich von Werten beschreiben wollen. Soll zum Beispiel geprüft werden, ob die Char-Variable *EingabeZeichen* einen der Werte 'A', 'a', 'X', 'x' hat, so können Sie den folgenden Ausdruck verwenden (es handelt sich bei der Aufzählung der Werte in eckigen Klammern um ein SET, eine Pascal-Menge, auf die in einem späteren Kapitel genauer eingegangen wird):

```
EingabeZeichen IN ['A', 'a', 'X', 'x']
```

Dieser Ausdruck ist gleichbedeutend mit

```
(EingabeZeichen = 'A') OR (EingabeZeichen = 'a') OR
(EingabeZeichen = 'X') OR (EingabeZeichen = 'x')
```

Syntax der IF-Anweisung

Auch wenn oder gerade weil sich die Beispiele fast wie Alltagssprache lesen, jedenfalls wenn Sie die englischen Wörter gegen ihre deutsche Entsprechung austauschen, gibt es reichlich Fußangeln. Das erste Problem taucht auf, weil der BEGIN-END-Rahmen um eine Anweisungsgruppe fehlen darf, wenn die Gruppe nur eine einzige Anweisung enthält. Das Beispiel von oben dürfen Sie in Pascal noch näher an der Alltagssprache formulieren:

```
IF AnzahlKopien < 100
  THEN
    Kopienpreis := 0.10
  ELSE
    Kopienpreis := 0.09;
Kopienkosten := AnzahlKopien * Kopienpreis;
```

Aber sobald mehrere Anweisungen zum Dann- oder Sonst-Teil gehören, muß der betreffende Teil mit BEGIN und END gerahmt werden, weil das Semikolon, das die Anweisungen trennt, sonst als Abschluß der gesamten IF-Anweisung interpretiert würde. Da mit ELSE keine neue Anweisung, sondern nur der nächste Teil der noch nicht beendeten IF-Anweisung anfängt, gilt außerdem die Regel:

Merkzettel

Vor ELSE steht nie ein Semikolon!

Schachtelung von Bedingungen

Syntaktisch können Sie Bedingungen kombinieren, so viel Sie mögen. Die Anweisungen im JA- oder NEIN-Teil einer IF-Anweisung dürfen ohne weiteres selbst IF-Anweisungen sein. Die Anweisungen sind dann geschachtelt wie ineinander gepackte Kästchen. Versuchen Sie, bevor Sie weiterlesen, selbst ein kleines Programmstück zu schreiben, das in Abhängigkeit vom Promillewert Verhaltensempfehlungen für Autofahrer ausgibt. Haben Sie es probiert? Hier eine mögliche Lösung (in die Rubrik „Promille > 0.8" lassen sich noch weitere Fallunterscheidungen – zum Beispiel für die absolute Fahruntüchtigkeit – einbauen):

```
IF Promille > 0
  THEN
    BEGIN
      IF Promille > 0.8
        THEN
          BEGIN
            WriteLn (' Unzulässig, einen Wagen zu fahren. ');
            WriteLn (' Nehmen Sie auf jeden Fall ein Taxi! ');
          END
        ELSE
          BEGIN
            WriteLn (' Fahrvermögen schon beeinträchtigt.');
            WriteLn (' Es wird dringend geraten, nicht selbst zu
                    fahren!');
          END;
    END
  ELSE
    BEGIN
      WriteLn (' Das Autofahren ist erlaubt!');
    END;
```

Wenn Sie es selbst probiert haben, sah Ihr erster Lösungsversuch sicherlich anders aus. In der Alltagssprache, an der man sich natürlich orientiert, gibt es selten ein „Sonst" oder voneinander abhängige Bedingungen. Vielleicht haben Sie sich deswegen verheddert, aber auch ein ziemlich anders aufgebautes Programm kann durchaus korrekt sein. Komplettieren Sie das Programm mit einer Eingabeprozedur für den Promillewert und testen Sie, ob es die Auskunft gibt, die Sie von ihm erwarten. Weiter un-

ten werden geschachtelte Bedingungen noch einmal zur Sprache kommen.

Programmablaufpläne und Struktogramme

Diagramme wie die beiden oben skizzierten, die recht anschaulich zeigen, in welcher Reihenfolge in einem Programm was passiert, werden Programmablaufpläne genannt, oft abgekürzt PAP(s). Sie heißen auch Flußdiagramme, weil sich an ihnen der Fluß eines Programms schrittweise verfolgen läßt.

PAP nach DIN

Die DIN-Norm 66001 legt fest, wie die Kästchen und Striche in einem PAP gemalt werden sollen. Soweit in diesem Buch überhaupt PAPs benutzt werden, seien gewisse Freiheiten gestattet. Im Standard-PAP symbolisiert das auf der Spitze stehende Quadrat (bzw. ein Rhombus), in dem eine Frage steht und das den Ausgangspunkt eines JA- und eines NEIN-Programmastes bildet, eine Verzweigung. Nach DIN-Norm müßte der JA-Ast geradeaus weitergehen und nur der NEIN-Ast zur Seite abbiegen.

Ein Rechteck steht für eine allgemeine Anweisung. Verbindungslinien bezeichnen die Reihenfolge des Ablaufs. Sie sind, außer wenn sie Pfeile mit der umgekehrten Richtung tragen, von oben nach unten und von rechts nach links zu lesen. Ein Oval ist eine Anfangs- oder Endstelle des Programmflusses.

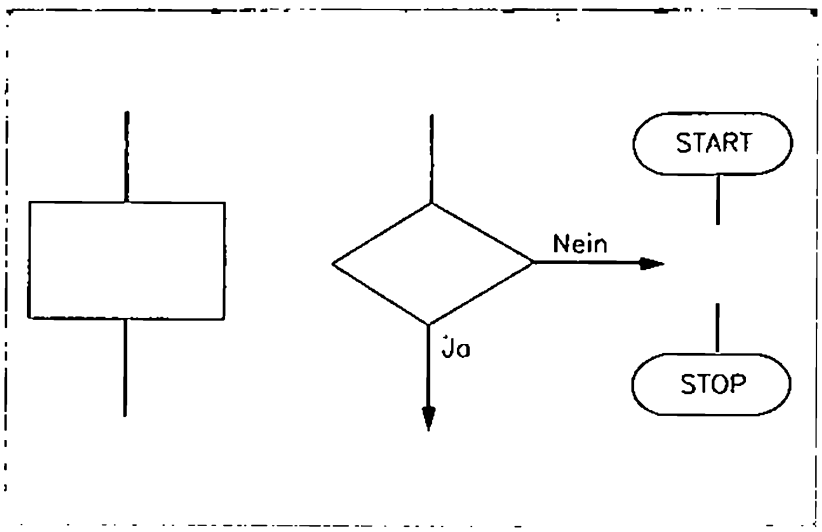

Abbildung 8-5: Wichtige Symbole des Programmablaufplans nach DIN 66001

Verzweigung und Entscheidungsstruktur

Die Verzweigung sagt an sich nichts darüber aus, wie der Weg nach ihr weitergeht. Derartige einfache Elemente spielen in alten Programmiersprachen eine große Rolle. Im Gegensatz zur Entscheidungsstruktur ist hier nicht festgelegt, daß die sich trennenden Wege auch wieder einander treffen müssen. Es darf jeder Pfad zu einem eigenen Ende finden, sich mit anderen Pfaden kreuzen oder eine Brücke zu einem anderen Teil des Programms schlagen. Die Diagramme, welche auf diese Art entstehen, wirken sehr viel abwechslungsreicher, als wenn Sie sich auf das strenge Schema der Entscheidungsstruktur beschränken. Aber Ästhetik und Logik stehen hier im Widerspruch. Diagramme und Programme mit beliebig verzweigten Wegen werden ganz leicht undurchschaubar und unkorrigierbar. Nur mit sehr großem Aufwand läßt sich herausfinden, woher ein Fehler im Gewirr der Wege eines sogenannten Spaghetti-Codes kommt. Sehen Sie sich das folgende, noch relativ harmlose, aber schwer zu analysierende Beispiel eines nicht wohlstrukturierten Programmstücks an:

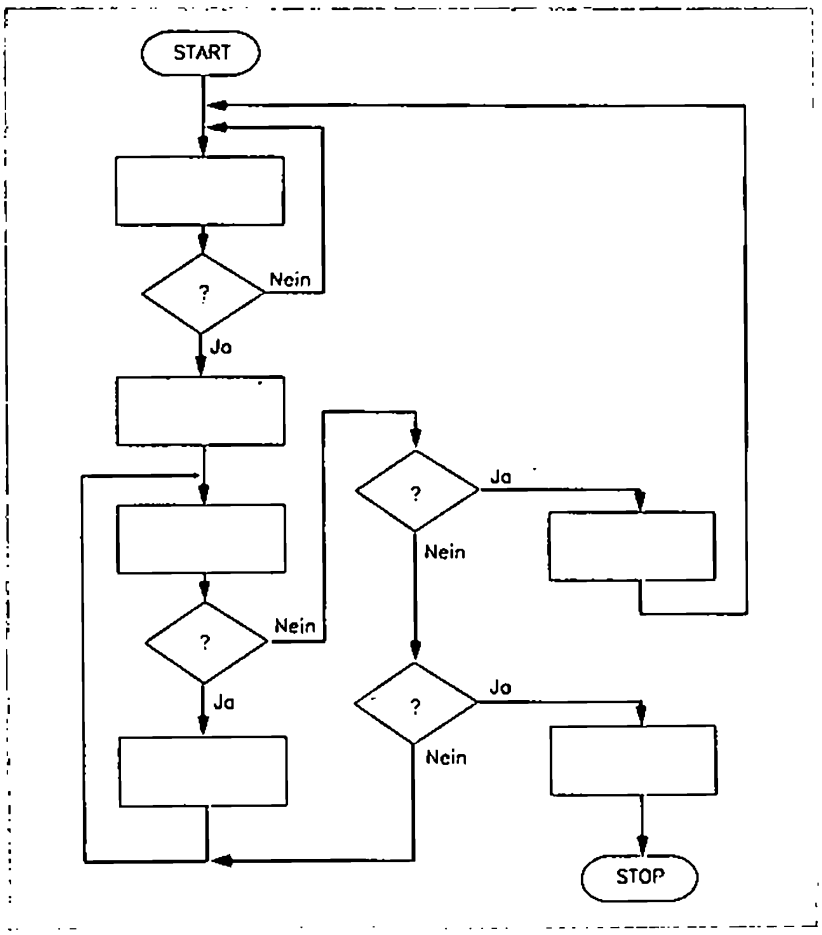

Abbildung 8-6: Schlecht strukturierter PAP

Nassi-Shneiderman-Diagramme

Eine richtige Entscheidungsstruktur führt nicht in ein Labyrinth. Jeder Pfad mündet wieder in den Hauptweg ein, jede Aufspaltung wird nach einer längeren oder kürzeren Strecke zurückgenommen. Einen Sprung rückwärts gibt es nicht. Damit von vornherein niemand mehr auf falsche Gedanken kommt, wurde eine Sorte von Diagrammen kreiert, die nach ihrem ganzen Aufbau keine Seiten- und Rücksprünge zuläßt. Nach ihren Schöpfern werden sie Nassi-Shneiderman-Diagramme oder auch Struktogramme (nicht zu verwechseln mit den Strukturdiagrammen) genannt. Bei diesen Diagrammen sind die Linien verschwunden, die an falsche Orte gezogen werden könnten. Die Kästchen für die einzelnen Anweisungen schließen auf voller Breite aneinander an, so daß Anweisungssäulen entstehen. Die Entscheidungsstruktur erscheint im Struktogramm als eine Aufteilung der Säule in eine linke und eine rechte Teilsäule.

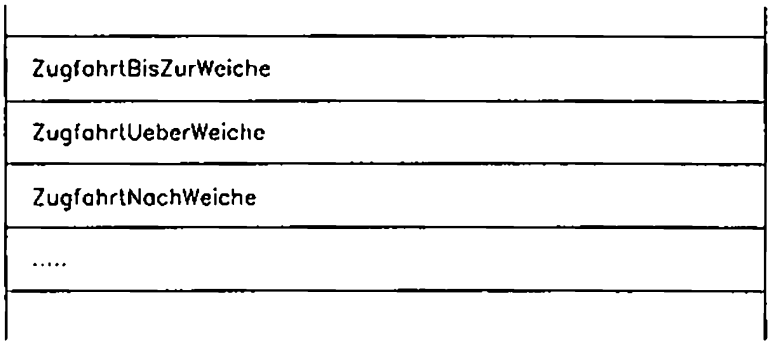

Abbildung 8-7:Sequentielle Struktur

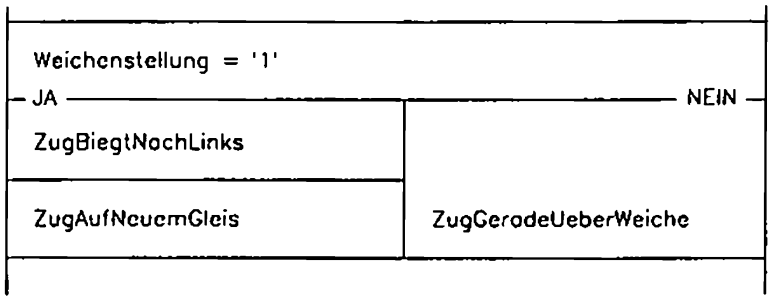

Abbildung 8-8: Entscheidungsstruktur

Faustregel zu Prozedurlänge

Als Faustregel gilt übrigens, daß ein einzelnes Struktogramm nie größer als eine DIN-A4-Seite sein soll. Wenn Sie merken, daß das Programm oder die Prozedur zu lang wird, ist es höchste Zeit, mehrere einzelne Anweisungen zu einer Prozedur zusammenzufassen. Im Hauptstruktogramm bleibt nur der Prozeduraufruf stehen.

Schrittweise Verfeinerung bei Struktogrammen

Um den Ablauf innerhalb der Prozedur zu beschreiben, verwenden Sie ein gesondertes Struktogramm, zu dem es eventuell wieder Unter-Struktogramme gibt und so weiter. Struktogramme sind also in derselben Weise hierarchisch aufeinander bezogen wie Prozeduren. Die Technik der schrittweisen Verfeinerung kann mit Struktogrammen genau abgebildet werden. In der alten Form der PAPs gibt es im Prinzip keine Darstellung von Unterprogrammen. Wenn das Blatt Papier nicht ausreicht, behilft man sich mit einem sogenannten Konnektor, einer Anschlußstelle. Das ist ein kleiner Kreis (oder auch eine Klammer) mit einer Kennzeichnung, zu der eine Verbindungslinie geführt wird. Er korrespondiert einem gleich gekennzeichneten Konnektor auf einem anderen Blatt.

Einseitige Entscheidungen und geschachtelte Strukturen

Eine einseitige Entscheidung ist beim Programmieren nicht die Bevorzugung einer Partei, sondern bedeutet, daß der ELSE-Teil einer IF-Anweisung nicht gebraucht wird. Für den Fall, daß die hinter IF stehende Bedingung nicht zutrifft, soll einfach nichts passieren. Es ist dann in Pascal erlaubt, den ELSE-Teil der IF-Anweisung entweder leer zu lassen oder auch gar kein ELSE zu schreiben. Das Struktogramm wird mit leerer rechter Teilsäule gemalt (oder zum Zeichen, daß Sie nicht einfach vergessen haben, etwas hineinzuschreiben, mit ./. markiert). Schematisch schreibt man:

```
IF Bedingungsausdruck        {Wenn ...}
  THEN                       {dann}
    BEGIN
      Anweisung1A;           {Eine oder mehrere}
      Anweisung1B;           {Anweisungen}
    END
  ELSE;                      {Der Sonst-Teil enthält nichts.}
```

Wenn ELSE weggelassen wird, steht das Schluß-Semikolon der IF-Anweisung schon nach dem Dann-Teil. Einseitige Entscheidungen sind zwar erlaubt, aber beim Programmieren mindestens mit der gleichen Vorsicht zu genießen wie in der Politik. Das folgende Schema „Wenn ... dann ..." ist ausgesprochen verführerisch:

```
IF Bedingungsausdruck
  THEN ...
```

Im Alltag argumentiert man mit auf bestimmte Fälle zugeschnittenen Ketten von Bedingungen. Es ist nicht notwendig, die „sonstigen" Fälle zu beschreiben, weil den Gesprächspartnern die Situation deutlich ist und das Gemeinte erschlossen werden kann. Aber der Computer versteht nicht, was man meint – er versteht gar nichts. Ein geflügeltes Wort der Softwarebranche lautet: „Der Computer tut, was wir ihm sagen – nicht, was wir von ihm wollen." Bei der IF-Anweisung und vor allem, wenn mehrere kombiniert angewandt werden, müssen Sie besonders aufpassen, daß das intuitive Sprachverständnis Ihnen nicht einen Streich spielt.

Empfehlung

Denken Sie stets bei der IF-Anweisung die Logik der Alternative zu Ende: nicht nur sagen, was geschehen soll, wenn irgendetwas sich so und so verhält – sondern auch die Konsequenzen im anderen Fall überlegen und dies im ELSE-Teil aufschreiben. Eine einseitige Entscheidung sollten Sie nur dann verwenden, wenn im anderen Fall tatsächlich nichts geschieht. Versuchen Sie nicht, mehrere Fälle mit getrennten, einzelnen IF-Anweisungen zu behandeln.

Umgang mit geschachtelten Bedingungen

Formal sind voneinander abhängige Entscheidungsstrukturen ganz einfach. Hauptsache, man schreibt sie übersichtlich auf, nach dem Muster:

```
Wenn Bedingung
  dann
    BEGIN
      Handlung1A
    END
  sonst
    BEGIN
      Handlung2A
        Wenn Bedingung2B
```

```
            dann  Handlung3A
          sonst Handlung4A
            Handlung2C
      END
```

Die Tiefe der Schachtelung darf dabei in verschiedenen Zweigen unterschiedlich sein. Im Struktogramm splitten sich manche Säulen auf, andere nicht:

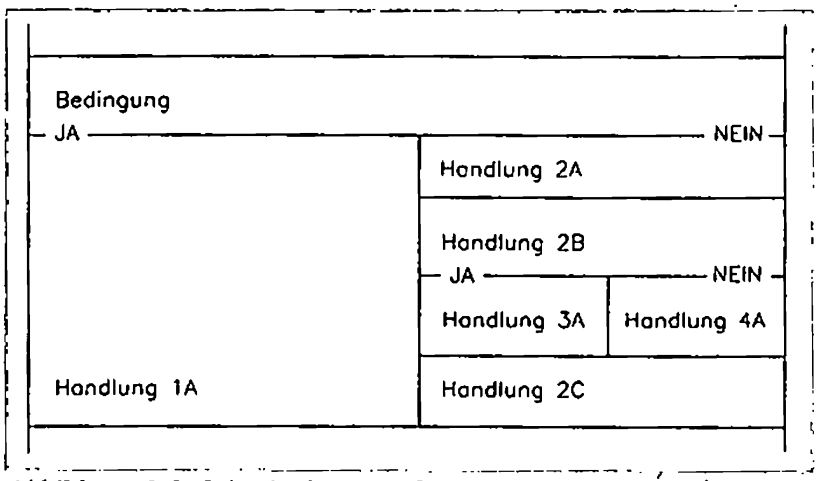

Abbildung 8-9: Schachtelung im Struktogramm

Oder betrachten Sie das Beispiel:

```
IF Temperatur > 25
  THEN
    BEGIN
      IF RegenAngesagt IN ['N','n']
        THEN
          BEGIN
            WriteLn (' Sachen zum Sonnen zusammenpacken');
            WriteLn (' in die Sonne legen    ');
          END
        ELSE
          BEGIN
            WriteLn (' Sachen zum Baden zusammenpacken ');
            WriteLn (' schwimmen gehen    ');
          END;
    END
  ELSE
    BEGIN
      IF RegenAngesagt IN ['N','n']
        THEN
          BEGIN
            WriteLn (' zum Spazierengehen umziehen  ');
            WriteLn (' fortgehen ');
```

```
        END
      ELSE
        BEGIN
         WriteLn ('zu Hause bleiben');
        END
  END;
```

Skizzieren Sie sich das Struktogramm hierzu.

Übersichtlicher mit Prozeduren

Die wichtigste Technik, um die Unübersichtlichkeit verwickelter Fallunterscheidungen zu vermindern, ist das Prinzip: reichliche Verwendung von Prozeduren, mit denen die Bedeutung von Programmabschnitten nachvollziehbar benannt wird. Auch das Beispiel von oben, an sich noch relativ kurz und klar, gewinnt an Verständlichkeit, wenn Sie die Prozeduren *FerienBeiWarmemWetter* und *FerienBeiKaltemWetter* einführen, denen Sie dann die Detailarbeit überlassen.

```
IF Temperatur > 25
  THEN
    FerienBeiWarmemWetter
  ELSE
    FerienBeiKaltemWetter;
```

Entscheidungs- und Wiederholungsbedingungen

Der Alltags-Gebrauch von „Wenn" führt beim Programmieren noch bei einer weiteren Situation in die Irre: Zum Beispiel hört sich die Aussage: „Wenn etwas falsch gelaufen ist (eine Tastatureingabe war unzulässig), dann muß es nochmal gemacht werden (erneut Taste drücken!)" ähnlich wie eine Entscheidungsstruktur an. Aber eigentlich soll ja dasselbe wie vorher nochmal versucht werden, nicht (bedingt) etwas Neues geschehen. Vor allem ist wahrscheinlich gemeint: Solange der Vorgang nicht korrekt ist, muß er *immer wieder* durchgeführt werden.

Ein Programm, in dem sich diese Vorstellung in IF-Anweisungen widerspiegelt, müßte theoretisch unendlich lang werden – obwohl eigentlich gar nichts Neues passiert. Die Situation läßt sich also mit IF-Anweisungen nicht beschreiben. Nicht alles, was als Bedingung formulierbar ist, kann vernünftig als eine Entscheidungsbedingung im Sinne der IF-THEN-ELSE-Anweisung interpre-

tiert werden. Es gibt eine zweite Sorte von Bedingungen, die Wiederholungsbedingungen genannt werden könnte. Um mit ihr umzugehen, wird eine neue Programmstruktur benötigt, die Wiederholungsstruktur. Sie bildet das Thema des nächsten Kapitels.

Aufgaben

1. Stellen Sie die IF-Anweisung mit Hilfe eines Syntaxdiagramms dar.
2. Versuchen Sie, ein Syntaxdiagramm für einen Bedingungsausdruck zu entwickeln.
3. Schreiben Sie das im letzten Beispiel angedeutete „Ferienratgeberprogramm" zu Ende.
4. Hier ist noch einmal der Gleisplan des Bahnhofs „Tuttlingen":

```
              /————————————-$     Gleis a
————————————-/————————————- Gleis b
——*————————/———————————— Gleis c
```

Abbildung 8-10: Gleisplan von Tuttlingen

Jetzt programmieren Sie die Weichenstellungen: Wo die Lok, die gerade auf Gleis c ankommt, letztlich landet, hängt von der Stellung der zwei Weichen ab. Schreiben Sie erst ein kleines Programm, bei dem die Stellung der Weichen eingegeben werden kann und daraus das Zielgleis ermittelt wird. Zeichnen Sie den zugehörigen PAP und das Struktogramm. Erweitern Sie Ihr altes Zugpro-

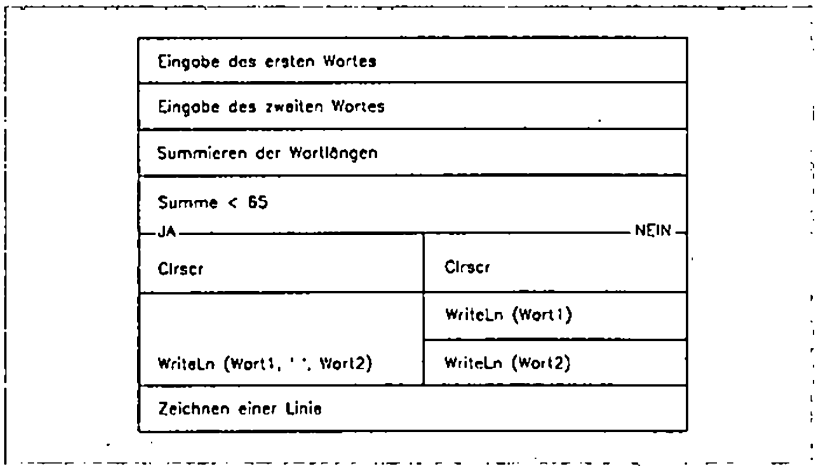

Abbildung 8-11: Aufgabe: Zeichnen Sie das Struktogramm als PAP.

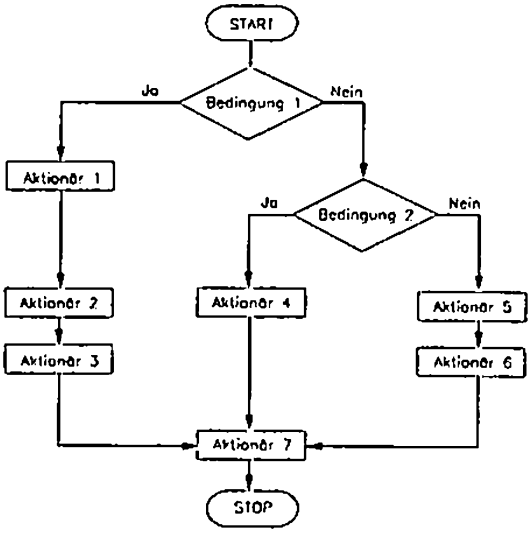

Abbildung 8-12: Aufgabe 6a): Zeichnen Sie den PAP als Struktogramm.

gramm so, daß die Lok auch auf dem Bildschirm richtig über die Weichen fährt.

5. Versuchen Sie zu erklären, was das durch das Struktogramm in Abb. 8-11 dargestellte Programmstück „macht". Zeichnen Sie den zugehörigen PAP.

6. Zeichnen Sie Struktogramme, bei denen das Programm dasselbe macht wie bei den PAPs in Abb. 8-12 und 8-13. Dabei dürfen beziehungsweise müssen eventuell Anweisungen im Struktogramm mehrfach verwendet werden – ein weiterer Anlaß, mehrfach vorkommende Anweisungsfolgen in Prozeduren zu verpacken.

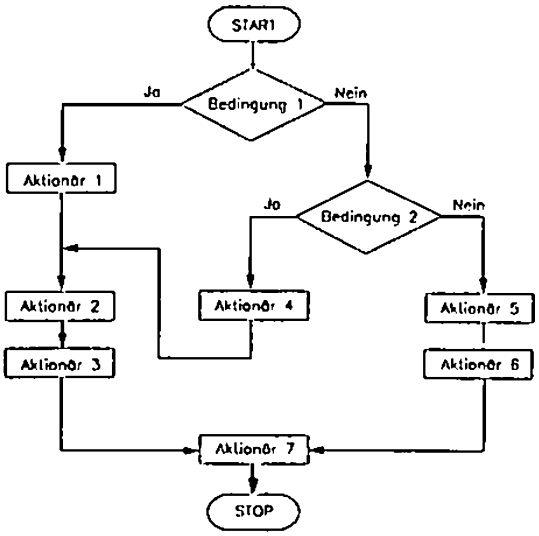

Abbildung 8-13: Aufgabe 6b): Zeichnen Sie den PAP als Struktogramm.

9 Wiederholungsstruktur und boolesche Variable

Der verpönte Sprung zurück – Die WHILE-Anweisung – Initial- und Finalzustände der Schleife – Boolesche Kontrollvariable für die Schleife – Die Verarbeitung des Zufalls – Ein kleines Würfelspiel – Simulation eines Ökosystems – Aufgaben

„Verweile doch, du bist so schön"
Goethe, Faust

Die Programmstruktur, die den Computer zur wiederholten Durchführung einer Aktion veranlaßt, heißt in der Programmiertheorie Wiederholungsstruktur. Gemeinsam mit der einfachen Sequenz (Folge) und der Entscheidungsstruktur bestückt sie den Baukasten für die Gestaltung des Programmablaufs im Kleinen.

Von den dreien ist die Wiederholungsstruktur das mächtigste und kritischste, mit ganz besonderer Vorsicht zu handhabende Mittel: Denn wehe, die vielen Schritte nehmen die falsche Richtung. Die Grafik symbolisiert die Wiederholungsstruktur als eine Kreisbewegung, die aber nur ein notwendiges und kontrolliertes Verweilen an einem Punkt des – vorwärts zielenden – Programmflusses bilden darf:

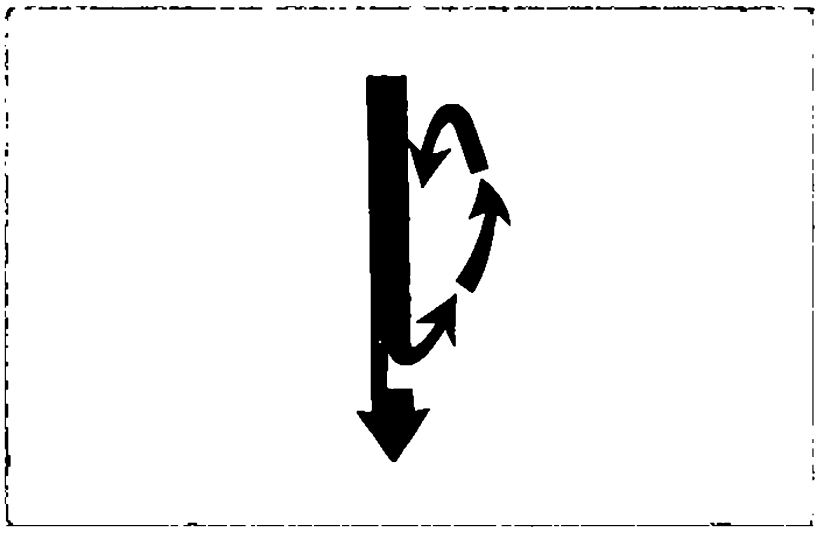

Abbildung 9-1: Eine Schleife im Programmfluß

Der verpönte Sprung zurück

Eigentlich ist nichts leichter, als den Computer dazu zu bringen, eine oder mehrere Anweisungen wiederholt auszuführen. Stellen Sie sich das Programm als eine Kette von Anweisungen vor. Soll ein Abschnitt nochmal durchgegangen werden, so muß nur hinter seiner letzten Anweisung zurück zu seinem Anfang gesprungen werden. Was dabei entsteht, wird anschaulich Programmschleife genannt. Im Regelfall darf die Schleife aber keine Endlosschleife sein, die nur durch einen äußeren Programmabbruch, im Extremfall mittels Stromentzug, gestoppt werden kann. Es muß einen ordentlichen Schleifenausstieg geben. Im klassischen DIN-Ablaufdiagramm wird eine Verzweigung verwendet, um einen sogenannten bedingten Sprung zurück zum Anfang der Schleife zu programmieren: Die *Anweisung_X* wird immer wieder durchlaufen, bis eine Bedingung (nicht mehr) erfüllt ist.

Auf Maschinenebene funktioniert letztlich eine Wiederholungsstruktur immer in dieser oder einer ähnlichen Form.

GOTO

Auch in den älteren Programmiersprachen wird nach diesem Schema verfahren. Um eine Schleife zu bauen, würden Sie dort eine Sprunganweisung verwenden, die in der Regel mit dem Wort GOTO gebildet wird. (GOTO für „Gehe zu", nicht zu verwechseln mit der Anweisung

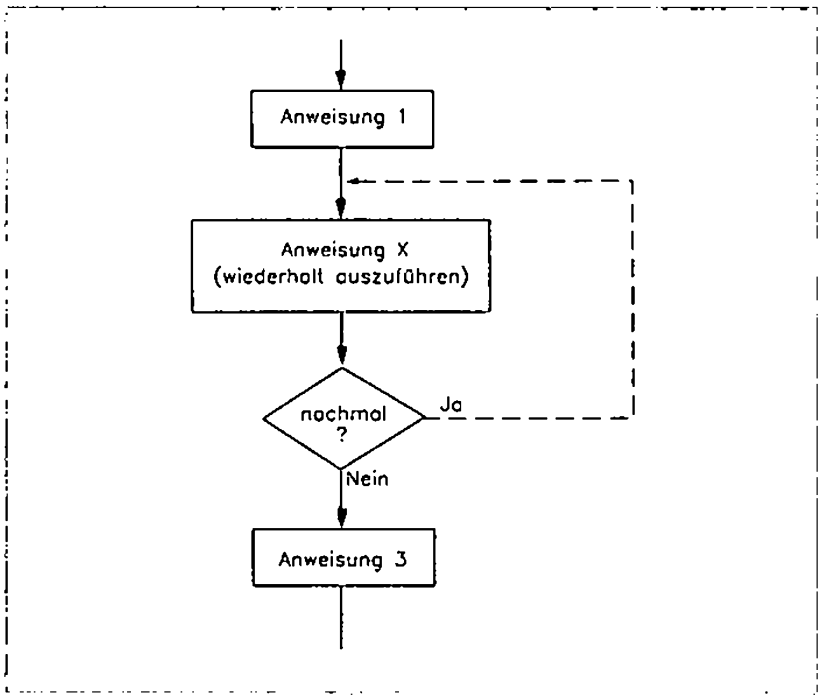

Abbildung 9-2: Wiederholung durch bedingten Sprung zurück

GoToXY, die nur für gezielte Bewegungen auf dem Bildschirm zuständig ist.)

Diese Sprunganweisung besitzt eine archaische Mächtigkeit, die nicht durch Regeln und Vorschriften eingegrenzt ist. Denn in der klassischen Form kann mit Sprunganweisungen die Kette der Programmanweisungen beliebig vernetzt werden. Mit GOTO kann man vorwärts, rückwärts, seitwärts springen, den Programmfluß hin- und zurückwenden sowie zwei Programmlinien eine Strecke lang ineinander flechten und dann versuchen, sie wieder zu trennen. Anstelle einer klar zu verfolgenden Linie des Programmablaufs entsteht ein vielfach verknotetes System von Anweisungsreihen. Es ist nicht besonders aufwendig, mit Hilfe von GOTO-Anweisungen ein Programm zu schreiben, das keine außenstehende Person, ja mit einiger Wahrscheinlichkeit nicht einmal mehr sein Autor versteht.

Das Beispiel für den Spaghetti-Code aus dem letzten Kapitel können Sie noch als harmlosen Fall betrachten. Das Riskante an so einem Code ist, daß ein Teil der Sprünge zurück führt. Seiten- und Vorwärtssprünge sind für sich nicht so problematisch, weil sie den Programmablauf nicht verlängern und nicht in Wiederholungen hineinführen. Sie können sich aber böse Fallen bereiten, wenn Sie bei Sprüngen (die übrigens mit mehr oder weniger starken Einschränkungen in Pascal sogar zugelassen sind) die Übersicht verlieren. GOTO ist gefährlich, weil Sie den Befehl einfach lokal an irgendeine Stelle setzen können, er aber globale Auswirkungen auf einen ganzen Zusammenhang von Anweisungen hat. In den heutigen Programmiersprachen verwendet man GOTO nur unter ganz bestimmten Umständen. In der Regel werden für spezielle Situationen eigene, eingegrenzte Anweisungsarten verwendet. In Pascal können Sie Strukturen aufbauen, mit denen eine Wiederholung ohne ausdrücklichen Rücksprungbefehl möglich ist.

Die WHILE-Anweisung

Bemerken Sie, daß ein Abschnitt des Programms, unmittelbar nachdem er ausgeführt wurde, in mehr oder weniger identischer Form noch ein oder mehrere Male benötigt wird, so identifizieren Sie ihn in Pascal als einen einzigen, komplexen Vorgang. Dieser ist – in einer

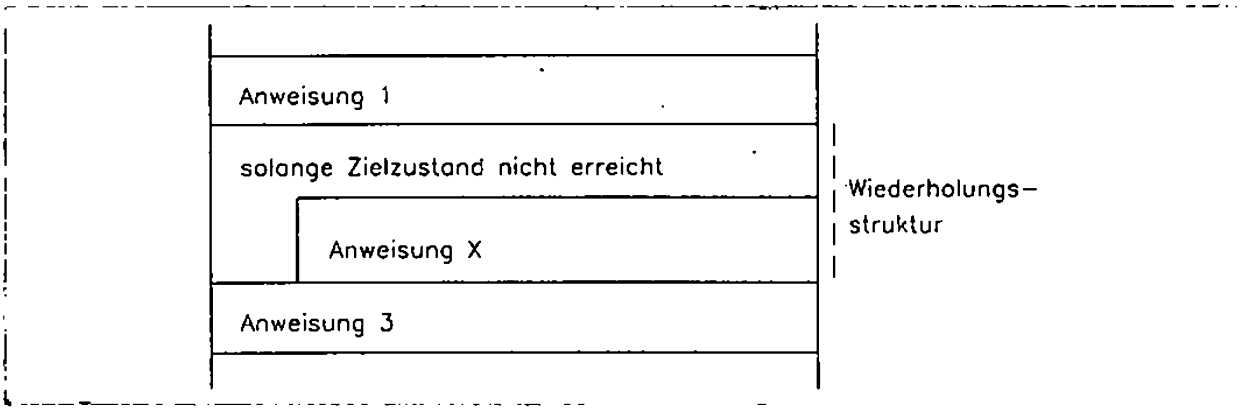

Abbildung 9-3: Struktogramm für die Wiederholungsstruktur mit WHILE

„Schleife", einer Wiederholungsstruktur – solange durchzuführen, bis ein vorgegebener Zielzustand erreicht ist. Im Hauptfluß des Programms steht nur die Anweisung, daß das Schleifenziel angestrebt wird. Ihr untergeordnet, durch sie begrenzt und kontrolliert, spielt sich die Wiederholung der inneren Anweisungsfolge ab.

Dieser Gedanke drückt sich im Struktogramm so aus, daß der Kontrollteil der Wiederholungsstruktur außen durchläuft, während die zu wiederholende Aktion einen eigenen, inneren Kasten (beziehungsweise eine innere, in Einzelrechtecke unterteilte Säule) bekommt. Im abgebildeten Schema gilt als zweite Anweisung des Programmflusses die Wiederholungsstruktur insgesamt.

Dem Struktogramm für eine Solange- oder WHILE-Schleife entspricht nicht ganz der PAP von oben, sondern

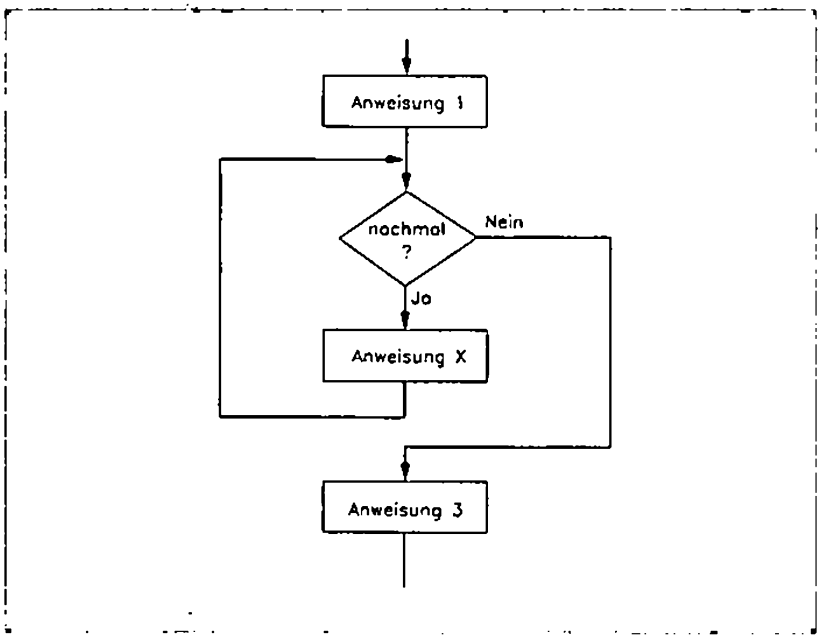

Abbildung 9-4: PAP zur Wiederholungsstruktur mit WHILE

ein PAP, bei dem die Verzweigung mit der Frage „ob es nochmal geschehen soll" vor den betreffenden Anweisungen steht.

Was passiert bei einer WHILE-Anweisung?

Der Ablaufplan zeigt, was geschieht: Zuerst prüft der Computer die Wiederholungsbedingung. Ist sie erfüllt, wird *Anweisung_X* ausgeführt, andernfalls zu *Anweisung_3* weitergegangen. Nach der Abarbeitung von *Anweisung_X* springt der Programmfluß zurück zur erneuten Abfrage der Wiederholungsbedingung.

Der PAP gibt den tatsächlichen Ablauf der Computeraktionen genau wieder. Aber er drückt mit keinem speziellen Symbol aus, welche Schleifenstruktur gemeint sein soll (bei einem konkreten Programm ist die Struktur in der Regel nicht so deutlich wie bei der Schemazeichnung). Die Verzweigung können Sie in den PAP setzen, sogar wenn Sie noch nicht wissen, ob Sie eine Entscheidungs- oder auf eine Wiederholungsstruktur bauen wollen. Jede Unklarheit führt aber zu fehlerhaften Programmen.

Schema der WHILE-Anweisung

In Pascal wird die in den Diagrammen dargestellte Programmstruktur mit einer WHILE-Anweisung erzeugt, die schematisch lautet:

```
Anweisung_1;
WHILE Bedingungsausdruck DO
                        {"Solange ... tue". Solange die Bedingung,
                        die hier steht, wahr ist,
                        wird in der Schleife verweilt.}
    BEGIN
      Anweisung_X;          {Eine oder mehrere Anweisungen}
    END;
Anweisung_3;
```

Ähnlich wie im THEN- und ELSE-Teil der IF-Anweisung darf der BEGIN-END-Rahmen hinter DO fehlen, wenn keine längere Anweisungsfolge, sondern nur eine Einzelanweisung wiederholt werden soll.

Das englische WHILE erinnert schon im Klang an das verwandte deutsche Wort „Verweilen" und trifft damit auch die Bedeutung „an einer Stelle bleiben, bis die Sache erledigt ist". Statt auf das Springen zwischen Einzelanweisungen ist das Augenmerk auf den Wiederholungsvorgang als ganzes gerichtet. Anders ausgedrückt: Die WHILE-Anweisung erzwingt im Gegensatz zu GOTO beim Niederschreiben eine globale Sicht.

Ein Beispiel zur WHILE-Anweisung

Betrachten Sie nochmal das Beispiel mit dem Alkoholspiegel im Blut. Das folgende Programm stellt den zeitlichen Verlauf des Promillewertes dar. Es beruht dabei auf der Abschätzung, daß pro Stunde etwa 0,1 Promille abgebaut wird. Solange der Wert noch nicht unter 0,1 gefallen ist, spürt man noch die Auswirkungen des vergangenen Abends: Man verweilt sozusagen noch im *Kater*. Der BEGIN-END-Rahmen um den Aufruf der Prozedur *Kater* ist an sich überflüssig, da zum DO keine weitere Anweisung gehört.

```
PROGRAM AlkoholAbbau;

USES crt, printer;

CONST
  PromilleGrenze = 0.8;
  Abbaurate      = 0.1;

VAR
  AnfangsPromilleWert   : Real;
  RestPromilleWert      : Real;
  VerstricheneStunden   : Integer;

PROCEDURE Kater;
  BEGIN
    ClrScr;
    GoToXY (1,24);
    WriteLn ('Bitte mit der Taste Return');
    WriteLn ('den Ablauf einer Stunde bestätigen.');
    GoToXY (1,3);
    WriteLn ('----------------------------------------------');
    IF RestPromilleWert > Promillegrenze
      THEN
        BEGIN
          WriteLn ('Keinesfalls Auto fahren! ');
          WriteLn ('Restalkohol in Ihrem Blut:'
                   ,RestPromilleWert:4:1, ' Promille')
        END
      ELSE
        WriteLn ('Lieber kein Auto fahren (noch'
                 ,RestPromilleWert:4:1, ' Promille).');
    WriteLn ('----------------------------------------------');
    WriteLn ('Anzahl der verstrichenen Stunde(n): ',
            VerstricheneStunden, '  Stunden');
```

```
    ReadLn:
    VerstricheneStunden := VerstricheneStunden + 1;
    RestPromilleWert    := RestPromilleWert - Abbaurate;
  END (* von Kater *);

BEGIN (* von Hauptprogramm *)
  ClrScr;
  WriteLn
        ('Dieses Programm ermittelt den Alkoholabbau pro Stunde.');
  WriteLn;
  Write   ('Geschätzter Promille-Wert an diesem Abend: ');
  ReadLn (AnfangsPromilleWert);
  WriteLn;
  VerstricheneStunden := 0;
  RestPromilleWert := AnfangsPromilleWert;
  WHILE RestPromilleWert >= 0.1 DO
    BEGIN
      Kater;
    END;
  ClrScr;
  GoToXY  (1,10);
  WriteLn ('Erst nach ', VerstricheneStunden,
          ' Stunden liegt Ihr Blutalkoholspiegel');
  WriteLn ('wieder unter 0.1 Promille.');
END.
```

Empfehlung zur Struktogramm-Gestaltung

Die Struktogramme zu diesem Beispiel sind getrennt für Hauptprogramm und Prozedur gezeichnet. Bei Verwendung von Prozeduren ist es sinnvoll, jedes Struktogramm zu rahmen und oben in den Rahmen den Namen der Prozedur beziehungsweise „Hauptprogramm" ein-

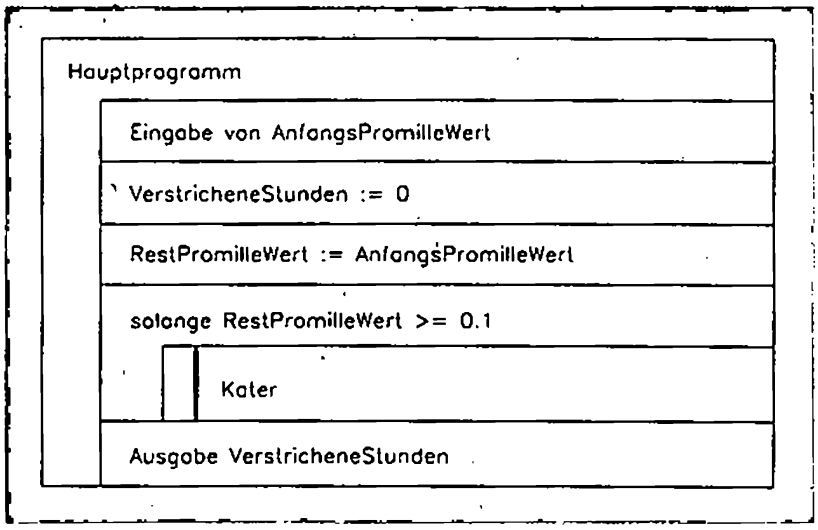

Abbildung 9-5: Struktogramm zum Hauptprogramm

zutragen. Außerdem sollte ein Prozeduraufruf von einer einfachen Anweisung durch eine besondere Kästchenform unterschieden werden (zum Beispiel mit einem seitlichen Doppelstrich).

Initial- und Finalzustände einer Schleife

Zur Abwechslung nun ein kleines Beispiel aus dem Leben eines fliegenden Händlers. Die Variablen *Erloes* und *TagesEinnahme* sollen als Real vereinbart sein. Die Prozedur *VerkaufEinerWare* wird definiert durch:

```
PROCEDURE VerkaufEinerWare;
  BEGIN
    WriteLn;
    WriteLn ('Verkaufen Sie noch etwas!');
    WriteLn;
    Write   ('Sie erhalten hierfür DM ');
    ReadLn  (Erloes);
    TagesEinnahme := TagesEinnahme
                     + Erloes;
  END (* von VerkaufEinerWare *);
```

Der folgende Abschnitt aus dem Programm *Haendler-Tag* bedeutet, daß der Verkauf so lange weiter laufen muß, bis die Tageseinnahmen mindestens DM 1000 betragen.

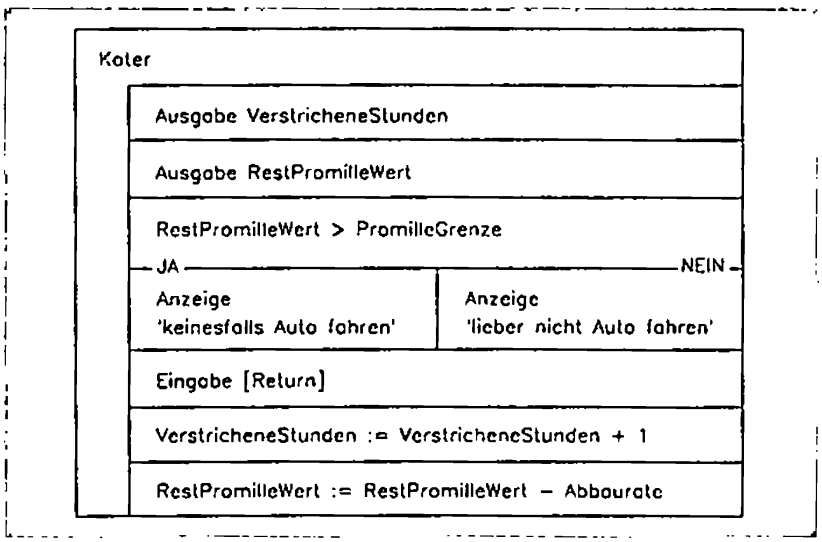

Abbildung 9-6: Struktogramm „Kater"

```
TagesEinnahme := 0;
WHILE TagesEinnahme < 1000 DO
    VerkaufEinerWare;
WriteLn;
WriteLn ('Endlich 1000 DM beisammen ',
        'genau sogar DM ',
          TagesEinnahme:8:2);
```

Schleife initialisieren

Zu einer Schleife gehört ein bestimmter Anfangs- oder Initialzustand. Es muß dafür gesorgt werden, daß eine oder mehrere Variablen auf die Werte gesetzt werden, welche die Grundlage für die Tätigkeit der Schleife bilden. Im letzten Beispiel wurde *TagesEinnahme* der Anfangswert 0 zugewiesen. Beim Programm *Alkoholabbau* mußte vor der Schleife dafür gesorgt werden, daß *RestPromilleWert* am Anfang den zutreffenden Wert enthält und *VerstricheneStunden* bei 0 zu zählen beginnt. Vergessen Sie die Initialisierung, so produziert die Wiederholungsstruktur falsche oder gar keine Ergebnisse, verweigert ihre Dienste oder verfängt sich in einer Endlos-Schleife.

Finalzustand (Zielzustand) definieren und erreichbar machen

Zweitens soll der End- oder Finalzustand einer Schleife klar definiert sein. Das weitere Programm kann dann auf der zuverlässigen Grundlage arbeiten, daß bei Abschluß der Schleife dieser Zustand vorgelegen hat. In den beiden Beispielen sind die Endzustände durch einfache Bedingungen charakterisiert, die (beziehungsweise deren logische Umkehrung) direkt im Bedingungsausdruck der WHILE-Anweisung stehen: *Tageseinnahme >= 1000* und *RestPromilleWert < 0.1*. Häufiger kommt es aber vor, daß die mit der Schleife angestrebten Ziele nicht so einfach zu beschreiben sind. Dazu gleich mehr.

Drittens darf die Schleife nicht zur Endlosschleife werden: Der Endzustand muß erreichbar sein. Die Anweisungen in der Schleife müssen gewährleisten, daß der Anfangszustand tatsächlich schrittweise in den Endzustand überführt wird.

Natürlich passiert manchmal ein Malheur, weil Umstände vergessen werden, die mindestens unter gewissen Bedingungen verhindern, daß das Ziel erreicht wird. Ein armer Händler könnte zum Beispiel von der Nacht überrascht werden, bevor er sein Tagespensum geschafft hat. Erweitern Sie deshalb das Handelsprogramm um die Kontrolle der Uhrzeit.

Boolesche Kontrollvariable für die Schleife

Endzustand der Schleife des Programms soll jetzt sein, daß entweder (das eigentlich gewünschte) Ziel einer Einnahme von mehr als 1000 DM erreicht ist oder eine Integer-Variable (der Einfachheit halber nur für die volle) Stunde den Wert 18 Uhr angenommen hat. Sie können dann einen Bedingungsausdruck aus den beiden Einzelbedingungen zusammenstellen.

Aber Vorsicht, die Bedingung, die bei WHILE steht, beschreibt nicht den Endzustand, sondern gibt an, wann die Schleife weiterlaufen muß. Der Händler muß so lange weiterarbeiten, wie es noch nicht 18 Uhr ist und er noch keine 1000 DM erwirtschaftet hat.

```
TagesEinnahme := 0;
Stunde        := 8;

WHILE (TagesEinnahme < 1000) AND (Stunde < 18) DO
  BEGIN
    VerkaufEinerWare;
    Write  ('Uhrzeit (Stunden): ');
    ReadLn (Stunde);
  END;
WriteLn;
IF TagesEinnahme >= 1000
  THEN
    WriteLn  ('1000 DM beisammen, genau DM ', TagesEinnahme:8:2)
  ELSE
    WriteLn ('Kein Verkauf mehr möglich,  Einnahme DM ',
             TagesEinnahme:8:2);
```

Formale Ableitung der Schleifenbedingung

Formal kann die Bedingung dafür, daß die Schleife weiter arbeiten soll, durch die logische Umkehr (die Negation oder Verneinung) einer Finalbedingung gewonnen werden. Der Händler hört auf, wenn:

```
(TagesEinnahme >= 1000) OR (Stunde >= 18)
```

Statt sich selbst zu überlegen, daß er demzufolge weitermacht, solange

```
(TagesEinnahme < 1000) AND (Stunde < 18)
```

können Sie diese Mühsal auch dem Pascal-System überlassen und einfach schreiben:

```
NOT ((TagesEinnahme >= 1000) OR (Stunde >= 18))
```

Boolean und die Alltagslogik

Pascal gibt Ihnen ein Mittel in die Hand, mit dem sich in komfortabler Weise nahe an der Alltagslogik programmieren läßt. Es sind die booleschen Variablen, die nur den Wert true oder false annehmen können. In anderen Programmiersprachen, die diese Variablentypen nicht kennen, könnten Sie stattdessen eine Signalvariable (Flag) nehmen und würden sie auf 1 beziehungsweise 0 setzen.

Mit booleschen Variablen können Sie Schleifen ohne großen formallogischen Aufwand steuern. Indem Sie dabei nahe am Alltagsverständnis einer Situation bleiben, wird Ihr Programm überschaubar und nach menschlichem Ermessen korrekt werden. Konkret: Sobald die Endbedingungen einer Schleife komplizierter werden, ist es dringend zu empfehlen, in einer eigenen booleschen Variable eine Notiz anzulegen, ob der Finalzustand erreicht ist. Diese Notiz muß immer auf den aktuellen Stand gebracht werden. Das erweiterte Händlerprogramm gewinnt zum Beispiel durch die Einführung einer booleschen Variable *MussNochArbeiten* erheblich an Klarheit. Die Schleifenkontrollvariable wird als Abschluß der Initialisierung auf „Finalzustand nicht erreicht" gesetzt. Bei jedem Schleifendurchlauf wird überprüft, ob die Endbedingung eingetreten ist.

Weil die Variable *MussNochArbeiten* selbst einen logischen Wert enthält, ist sie schon für sich ein Bedingungsausdruck und kann hinter WHILE alleine stehen (*MussNochArbeiten = true* würde dasselbe bedeuten).

```
TagesEinnahme     := 0;
Stunde            := 8;
MussNochArbeiten := true;

WHILE MussNochArbeiten DO
  BEGIN
    VerkaufEinerWare;
    IF TagesEinnahme >= 1000
      THEN MussNochArbeiten := false
    ELSE
      BEGIN
        Write ('Uhrzeit (Stunden): ');
        ReadLn (Stunde);
        IF Stunde >= 18
          THEN MussNochArbeiten := false;
      END;
```

```
  END;
WriteLn;
IF TagesEinnahme > 1000
  THEN
    WriteLn ('1000 DM beisammen, genau DM ', TagesEinnahme:8:2)
  ELSE
    WriteLn ('Kein Verkauf mehr möglich, Einnahme DM ',
            TagesEinnahme:8:2);
```

Zeichnen Sie sich das Struktogramm zu diesem Programm! Sie haben hier wieder eine Entscheidungsstruktur im Innern einer Wiederholungsstruktur.

In der neuen Form des Programms können die verschiedenen Situationen, unter denen der Schleifenabbruch erfolgt, genauer erfaßt werden. Zum Beispiel braucht die Uhrzeit gar nicht mehr festgestellt werden, wenn das Geld beisammen ist und die Arbeit schon deshalb gestoppt werden kann. Außerdem müssen Sie sich auf der Kontrollebene der Wiederholungsstruktur nicht mit den Details des Inneren befassen. Die Abbruchbedingungen könnten sich ja noch viel verwickelter gestalten. Zahlreiche weitere Variablen könnten mitspielen, auch die richtige Folge ihrer Werte könnte entscheidend sein. All diese Informationen brauchen nicht auf die äußere Ebene getragen zu werden. Sie stören und verwirren nur, wo es bloß auf das Ergebnis, „Schleife soll weiterlaufen" oder „Schleife soll nicht weiterlaufen", ankommt.

Der Rest des Kapitels benutzt die Schleifenprogrammierung, um mit dem Computer etwas Schicksal zu spielen.

Die Verarbeitung des Zufalls

Eine seiner wichtigsten Anwendungen findet der Computer in einem Bereich, in dem in gewissem Sinn der Zufall regiert: in der Simulation von Prozessen, die statistischen Gesetzen gehorchen. Simuliert werden solche Prozesse, weil es zu aufwendig, zu teuer, zu gefährlich oder ganz unmöglich wäre, sie in der Wirklichkeit zu beobachten. Wesentliches Element dessen, was sich bei ihnen wiederholt, ist ein zufälliges Ereignis wie der Fall eines Würfels.

Würfelsimulation mit Random

Vor einem Wurf weiß man nicht, welche der sechs Würfelseiten nachher oben liegt. Man kann aber davon ausgehen, daß jede Würfelseite die gleiche Chance hat, nach oben zu kommen. Das läßt sich sogar in Zahlen ausdrücken: Man sagt, 1/6 (oder 16.66 Prozent) beträgt die Wahrscheinlichkeit, daß beim Würfeln zum Beispiel die 5 herauskommt.

Die Grundlage für alle Simulationen mit dem Computer sind spezielle Hilfsfunktionen, die zufällige Werte aus einem Spektrum gleich wahrscheinlicher Zahlen abrufen. Zum Beispiel imitieren Sie einen Würfel mit einer Prozedur, die Ihnen mit gleicher Wahrscheinlichkeit Zahlen zwischen 1 und 6 liefert. In Turbo-Pascal gibt es zur Erzeugung zufälliger ganzer Zahlen die Funktion Random (englisch für „Zufall"), hinter der in runden Klammern die Länge des (bei 0 beginnenden) gleich wahrscheinlichen Zahlenspektrums steht. Die Länge vermindert um 1 ergibt dann die maximal herauskommende Zahl. Zum Beispiel steht *Random (6)* für einen zufällig aus 0 bis 5 herausgegriffenen Wert.

Mit Hilfe dieses zufälligen Wertes können Sie sich eine Prozedur zum „Würfeln" einer Zahl schreiben. Da Sie für einen Würfel nicht Zahlen zwischen 0 und 5, sondern zwischen 1 und 6 brauchen, verschieben Sie das von *Random (6)* produzierte Spektrum durch die Addition von 1. *Wuerfelzahl* ist dabei eine im Hauptprogramm deklarierte Integer-Variable.

```
PROCEDURE Wuerfeln;
  BEGIN
    Wuerfelzahl := Random (6) + 1;
  END (* von Wuerfeln *);
```

Die Random-Funktion wird auch ein Zufallsgenerator oder eine Zufallsfunktion genannt. Im Gegensatz zu Zufallsgeneratoren, die ihre Zahlen aus statistisch-physikalischen Vorgängen, zum Beispiel Molekülbewegungen, gewinnen, kommen die von Random produzierten Werte nicht wirklich zufällig zustande, sondern werden durch ein ausgetüfteltes Verfahren berechnet. Solche Zahlen heißen deshalb auch Pseudo-Zufallszahlen. Tatsächlich nicht vorherbestimmt ist hierbei eigentlich nur, welchen Wert man innerhalb der eindeutigen Abfolge als ersten erwischt. Um Programme besser testen zu können, kann diese Stelle (ab Turbo 4.0) fixiert werden. Umgekehrt können Sie auch verhindern, daß ein Programm immer wie-

der an derselben Stelle der Zahlenfolge beginnt: Ein Aufruf der vordefinierten Prozedur *Randomize* (randomisieren, „zufällig machen") wirft die Zahlenfolge in nicht kontrollierbarer Form durcheinander.

Ein kleines Würfelspiel

Als kleines Spiel können Sie mit der Prozedur *Wuerfeln* eine Art Wettrennen zwischen zwei Partnern, genannt A und B, programmieren: Beide würfeln abwechselnd beziehungsweise lassen den Computer würfeln. Wer zuerst 30 Punkte erreicht, hat gewonnen. 30 ist als Konstante *Ziel* definiert, außer *Wuerfelzahl* gibt es *SummeA* und *SummeB* als Integer-Variablen, *SpielLaeuft* ist die boolesche Schleifenkontrollgröße. Das Hauptprogramm könnte dann so aussehen:

```
PROGRAM Wuerfelspiel;

USES crt, printer;

CONST
  Ziel = 30;
  SpalteA    =  2;
  SpalteB    = 42;
  SpalteInfo = 15;

VAR
  Wuerfelzahl      : Integer;
  SummeA, SummeB   : Integer;
  SpielLaeuft      : Boolean;

PROCEDURE ProgrammErklaeren;
  BEGIN
    ClrScr;
    WriteLn;
    WriteLn ('Herzlich Willkommen zum Würfelspiel!');
    WriteLn ('Anzahl der Mitspieler: 2');
    WriteLn
      ('Spielregel: Gewonnen hat, wer zuerst 30 Punkte erwürfelt.');
    Write   ('Start und Abschluß des Wurfes mit der Taste',
             '[RETURN].');
    WriteLn;
    Write ('----------------------------------------');
    Write ('----------------------------------------');
    WriteLn;
    WriteLn;
  END;
```

```
PROCEDURE Wuerfeln;
  BEGIN
    Wuerfelzahl := Random (6) + 1;
  END (* Wuerfeln *);

PROCEDURE ErgebnisMitteilung;
  BEGIN
    WriteLn;
    WriteLn ('———————————————————',
            'Ergebnis ———————————————————');
    WriteLn;
    IF (SummeA >= Ziel) AND (SummeB >= Ziel)
      THEN
        BEGIN
           WriteLn ('Unentschieden, weil A und B in der ');
           WriteLn ('gleichen Runde über 30 gekommen sind!')
        END
      ELSE                    {A oder B hat Ziel nicht erreicht.}
         BEGIN
           IF SummeA >= Ziel
                                {Hat A das Ziel 30 erreicht?}
             THEN
               WriteLn ('Spieler(in)  A  hat gewonnen! ')
             ELSE        {Wenn A nicht am Ziel, muß B angekommen
                         sein, sonst würde das Spiel noch laufen.}
               WriteLn ('Spieler(in)  B  hat gewonnen! ');
         END;
  End; (* Ergebnismitteilung *)

BEGIN (* KleinesWuerfelspiel *)
  ProgrammErklaeren;
  ReadLn;
  SummeA := 0;
  SummeB := 0;
  SpielLaeuft := true;
  Randomize;
  ClrScr;
  WHILE SpielLaeuft DO
    BEGIN
      GoToXY  (SpalteInfo,24);
      WriteLn ('Bitte die Taste [RETURN] drücken!');
      GoToXY  (SpalteInfo,8);
      Write   ('An der Reihe ist Spieler(in):   A');
      GoToXY  (SpalteA+20,9);
      ReadLn;
      Wuerfeln;
```

```
      SummeA: = SummeA + Wuerfelzahl;
      GoToXY  (SpalteA,2);
      Write   ('Gewürfelt          :   ', Wuerfelzahl);
      GoToXY  (SpalteA,4);
      Write   ('Summe Spieler(in) A: ',  SummeA : 3);
      GoToXY  (SpalteInfo,8);
      Write   ('An der Reihe ist Spieler(in):   B');
      GoToXY  (SpalteB+20,9);
      ReadLn;
      Wuerfeln;
      SummeB := SummeB + Wuerfelzahl;
      GoToXY  (SpalteB,2);
      Write   ('Gewürfelt          :   ', Wuerfelzahl);
      GoToXY  (SpalteB,4);
      Write   ('Summe Spieler(in) B: ',  SummeB : 3);
      IF (SummeA >= Ziel) OR (SummeB >= Ziel)
        THEN SpielLaeuft := false;
    END;
  ClrScr;
  ErgebnisMitteilung;
END.
```

Summenbildung in einer Schleife

Das Programm zeigt die für viele EDV-Anwendungen typische Verwendung einer Summenvariablen im Zusammenhang mit einer Schleife. Die Variable wird vor Beginn der Schleife, in der Schleifeninitialisierung, auf 0 gesetzt. Nach der Schleife steht auf der Variablen dann die Gesamtsumme der Werte.

Simulation eines Ökosystems

Zum Schluß des Kapitels noch ein kleines Programm, das die Wirkungen des Einsatzes von Pestiziden statt in der Wirklichkeit versuchsweise im Modell eines Ökosystems simuliert. Das Modell muß ein vernetztes System abbilden, in dem die Änderung einer Größe oft unüberschaubare Rückwirkungen auf andere Größen hat.

Qualitative Modellvorstellung

Zugrunde gelegt wird ein abgewandeltes „Räuber-Beute-Modell": Ein landwirtschaftlicher „Schädling", zum Beispiel die Spinnmilbe, pflanzt sich mit einer hohen Geburtenrate fort. (In der Natur selber kann man nicht nach „Schädlingen" und „Nützlingen" sortieren. Der Begriff findet nur der Einfachheit halber Verwendung). Eine zweite Tierart, zum Beispiel Raubmilben, ernährt sich

„räuberisch", weil sie Schädlinge frißt. Sie kann daher um so mehr Nachkommen aufziehen beziehungsweise erzeugen, je zahlreicher es den Schädling gibt. Unter natürlichen Bedingungen stellt sich in der Regel ein Gleichgewicht ein: Wird die Schädlingspopulation zu groß, vermehren sich auch die von ihr lebenden Räuber stark. Wenn sie die Beute dezimiert haben, wird ihre Nachkommenschaft kleiner, so daß ein Teil der Schädlinge übrig bleibt. Damit ist der künftige Bestand der Räuber garantiert. Insgesamt pendelt sich ein ungefähres Gleichgewicht zwischen den beiden Tierarten ein.

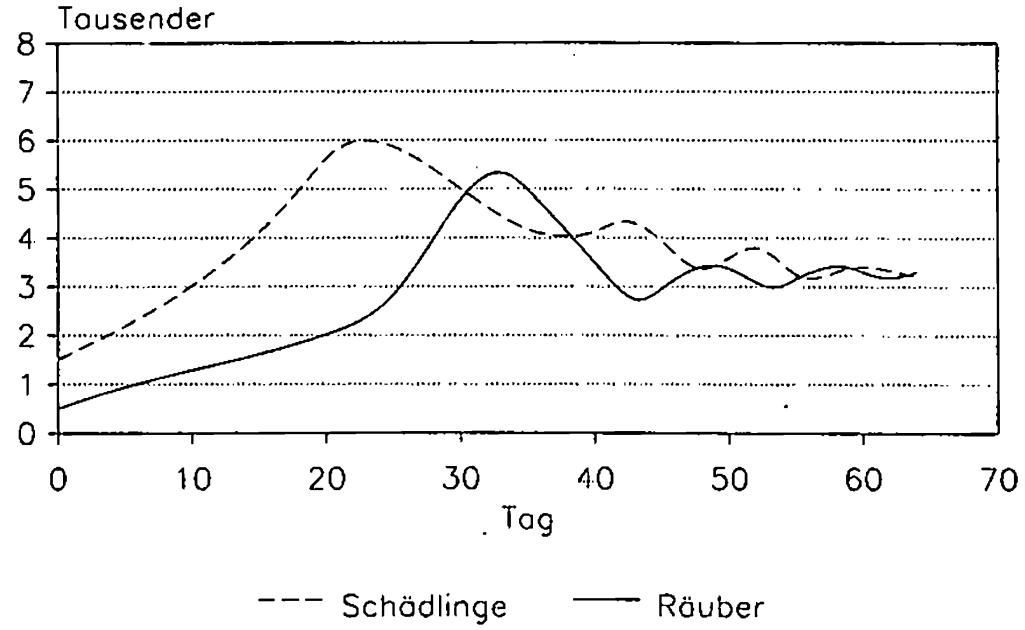

Abbildung 9-7: Ein Ökosystem strebt ins Gleichgewicht.

Greift der Mensch mit der Anwendung eines Giftes ein, das zunächst beide Tierarten gleichmäßig trifft, so erholen sich die Schädlinge aufgrund ihrer hohen Vermehrungsrate sehr viel schneller als die Räuber, die eventuell sogar ganz ausgerottet werden. Solche modellhaften Zusammenhänge versucht das Simulationsprogramm quantitativ wiederzugeben. Der Sinn einer Simulation besteht indessen weniger in einer exakten, quantitativen Voraussage – dazu stecken viel zu viele idealisierte Annahmen im Programm. Mit Hilfe eines solchen Programms läßt sich aber die Rolle verschiedener Einflußgrößen (den sogenannten Parametern des Modells, die im Programm durch Konstanten dargestellt sind) untersuchen: Vermehrungsraten, Revierdichten der Räuber etc.

Zur Realisierung

Im folgenden Programm wird angenommen, daß ein abgegrenztes Gelände vorhanden ist, das aus einer bestimmten Zahl (*AnzahlReviere*) von Räuberrevieren besteht. Die *S*-Tiere (die „Schädlinge") verteilen sich gleichmäßig über die Reviere (so daß jedes *R*-Tier im wesentlichen die Zahl von *SImRevier* = *BestandS / AnzahlReviere* als potentielle Opfer hat). Nicht jeder Versuch eines R-Tiers, ein Beutetier in seinem Revier zu erwischen, gelingt. Nur mit der allerdings ziemlich hohen *Erfolgswahrscheinlichkeit* ist dem Räuber das Jagdglück hold. Außerdem startet ein R-Tier höchstens *MaximaleJagdzahl* Versuche am Tag, ein S-Tier zu fangen. Wenn die Zahl der Schädlinge sehr hoch ist, liegt daher die Zahl *SGejagtImRevier* (der tatsächlich gejagten) unter der Zahl der vorhandenen S-Tiere, von denen dann viele unbehelligt bleiben. Die Schädlinge haben eine feste *VermehrungsrateS* pro Tag. Die Räuber können je verzehrtem S-Tier einen Nachkommen erzeugen. Bei Anwendung des Pestizids schrumpfen beide Tierpopulationen mit dem selben Faktor, dem *Ueberlebensfaktor*.

Vereinfachungen

Das ganze Modell ist natürlich in vielen Einzelheiten grob vereinfacht. Es wirken im wesentlichen keine anderen äußeren Einflüsse auf den Tierbestand als die genannten. Das Pestizid führt gleichmäßig zu einer Verminderung beider Tierarten auf einen festen Prozentsatz (in Wirklichkeit werden dank ihrer schnelleren Vermehrung die „Schädlinge" relativ häufig resistent, so daß das Gift nachher fast nur noch die „Nützlinge" trifft). Bei genauem Hinsehen verkürzt also die Computersimulation die Wirklichkeit: Allein schon deshalb, weil der Computer mit präzisen Werten und Größen in multikausalen und „offenen Systemen" rechnet. Er simuliert bloß eine eindimensionale Genauigkeit, wo Mehrdeutigkeit angemessen ist.

Simulation kein Ersatz für Argumentation

Für Computersimulationen gilt, daß ihr Ergebnis in der Realität nur so beweiskräftig ist, wie die Interpretation gewissenhaft und glaubwürdig durchgeführt wird. Über alle zweifelhaften Stellen muß sorgfältig Rechenschaft abgelegt werden. Eine Simulation kann nie die Beobachtung der Wirklichkeit ersetzen und eigentlich keine positive Urteile liefern, sondern letztlich nur warnen: Wenn etwas im Modell schief läuft, sollte man in der Wirklichkeit besonders vorsichtig operieren.

Wiederholungsstruktur

```
PROGRAM Oekosystem;

USES crt, printer; {Muß erst ab den Versionen
                    Turbo-Pascal 4.0 eingefügt werden.}

CONST
  EscapeTaste        = #27;
  AnfangsbestandS    = 1000.0;
  AnfangsbestandR    = 10.0;
  Ueberlebensfaktor  = 0.01;
                       {Bei Pestizidanwendung}
  AnzahlReviere      = 1000;
                       {Maximalzahl R, die nicht auswandern}
  MaximaleJagdzahl   = 50; {So viel kann ein R maximal an einem Tag
                       zu jagen versuchen.}
  VermehrungsrateS   = 1.5;{Am Tag}
  SterbeWahrscheinlichkeitR = 5;
                       {Am Tag in ganzzahligen Prozent}
  Erfolgswahrscheinlichkeit = 70;
                       {In ganzzahligen Prozent Wahrscheinlichkeit,
                        daß ein R ein gejagtes S wirklich erwischt.}

VAR
  Tag                      : Integer;
  BestandS, BestandR       : Real;
  TagLaeuft                : Boolean;
  Zeichen                  : Char;
  RExemplar, SExemplar     : Integer;

PROCEDURE Zierzeile;
  VAR
    i: Integer;
  BEGIN
    FOR i:= 1 TO 80 DO Write ('*');

  END;

PROCEDURE Vermehrung;
  VAR
    AlterBestandR              : Integer;
    SImRevier, SGejagtImRevier : Integer;

  PROCEDURE Jagd;
  VAR
    Jagderfolg : Boolean;
  BEGIN                  {Das RExemplar jagt SExemplare in seinem
                          Revier.}
    FOR SExemplar := 1 TO SGejagtImRevier DO
```

```
    BEGIN
      IF Random (100) < Erfolgswahrscheinlichkeit
        THEN Jagderfolg := true
        ELSE Jagderfolg   := false;
      IF Jagderfolg
        THEN          {Wenn ein R Erfolg hat, kommt ein
                      Nachkomme zustande, ein S geht zugrunde.}
          BEGIN
            BestandR := BestandR + 1;
            BestandS := BestandS - 1;
          END;
    END;
  END (* von Jagd *);

BEGIN (* von Vermehrung *)
  IF BestandS / AnzahlReviere > MaxInt
    THEN
      SImRevier := MaxInt {Schutz gegen Programmabsturz bei zu
                           großen Zahlen.}
    ELSE
      SImRevier := trunc (BestandS / AnzahlReviere);
                           {Die S verteilen sich gleichmäßig auf
                           die Reviere.}
  IF SImRevier >= MaximaleJagdzahl
    THEN
      SGejagtImRevier  := MaximaleJagdzahl
    ELSE
      SGejagtImRevier  := SImRevier;
  IF BestandR > AnzahlReviere
    THEN
      BestandR := AnzahlReviere;
                           {Gibt es mehr Räuber als Reviere, so
                           wandern Tiere aus.}
  AlterBestandR := trunc (BestandR);
                           {Übertragen auf Integer-Größe und Merken
                           des Wertes}
  FOR RExemplar := 1 TO AlterBestandR DO Jagd;
                           {Jedes Räubertier jagt in seinem Revier.}
  BestandS := BestandS * VermehrungsrateS;
                           {Natürliche Vermehrung der S-Tiere ist
                           heftig, ihre Nahrung unbegrenzt.}
  AlterBestandR := trunc (BestandR);
  FOR RExemplar := 1 TO AlterBestandR DO
    IF Random (100) < SterbewahrscheinlichkeitR
      THEN
```

```
          BestandR := BestandR -1;
                                {Jedes Exemplar von R stirbt mit einer
                                gewissen Wahrscheinlichkeit.}
END (* von Vermehrung *);

PROCEDURE EinTagImOekosystem;
  BEGIN
    Tag := Tag + 1;
    Write ('Tag ',Tag:3);
    Write ('|| ');
    Vermehrung;
    Write (' Pestizidanwendung j/n ');
    Zeichen := ReadKey;    {In Turbo-Pascal 3.x muß diese Zeile
                            lauten: Read (kbd, Zeichen);}
    IF Zeichen = 'j'
      THEN
        BEGIN
          Write ('j');
          BestandS := Ueberlebensfaktor * BestandS;
          BestandR := Ueberlebensfaktor * BestandR;
        END
      ELSE
      IF Zeichen = 'n'
        THEN
          Write ('n')
        ELSE Write (' ');
    Write ('|');
    Write (' Anzahl S', BestandS : 15:0);
    Write ('|');
    Write (' Anzahl R', BestandR : 10:0);
    WriteLn;
  END (* von EinTagImOekosystem *);

BEGIN (* von Hauptprogramm *)
  ClrScr;
  WriteLn ('   Simulation eines  Räuber-Schädlings-Systems',
           '   Beenden mit Esc');
  Zierzeile;
  WriteLn ('  Anfangsbestand „Schädlinge" ', AnfangsBestandS:10:0);
  WriteLn ('  Anfangsbestand „Räuber"     ', AnfangsBestandR:10:0);
  WriteLn;
  WriteLn;
  Zierzeile;
  Randomize;
  BestandR := AnfangsBestandR;
  BestandS := AnfangsBestandS;
  Tag := 0;
```

```
  TagLaeuft := true;
  WHILE TagLaeuft
    DO
      BEGIN
        EinTagImOekoSystem;
        IF Zeichen = EscapeTaste
          THEN
            TagLaeuft := false;
      END;
  Zierzeile;
END.
```

Wenn Sie die Ergebnisse der Simulation grafisch auswerten oder in anderer Form nach dem Durchlauf des Programms nutzen möchten, so müssen Sie per Hand mitschreiben, die Werte ausdrucken lassen oder die Werte auf Diskette oder Platte abspeichern. Im nächsten Kapitel erfahren Sie, wie Sie in Pascal den Computer dazu veranlassen, beliebige Daten in Textdateien zu schreiben und sie wieder zu lesen.

Aufgaben

1. Zeichnen Sie das Syntaxdiagramm für die WHILE-Anweisung.

2. Damit WHILE-Schleifen zu einem Ende kommen, muß die von ihnen kontrollierte Aktion dazu führen können, daß die Eintrittsbedingung nicht mehr zutrifft. Was ist unter diesem Gesichtspunkt zu folgenden Ideen für Schleifen zu sagen:

```
WHILE MehrWissenWollen DO Fragen;
WHILE hungrig DO Essen;
WHILE virulent DO Infektion;
WHILE muede DO Schlafen;
WHILE Mehr DO Zugabe;
WHILE Again DO TryIt;
```

3. Das folgende Struktogramm (Abb. 9-8) beschreibt ein Programm, mit dem ein kleiner Laden jeden Tag sein Kassenbuch führt. Am Morgen ist der Anfangsbestand der Kasse einzutippen. Ein hinzukommender oder abgehender Betrag wird registriert. Auf Wunsch kann ein Zwischenabschluß anzeigt werden. Das Ganze wieder-

holt sich, solange *Weiterarbeiten* bejaht wird. Am Ende wird ein Tagesabschluß erstellt.

4. Finden Sie die WHILE-Schleife, die dasselbe tut, wie die folgende Zählschleife, wobei die Variablen *i*, *Anfangswert*, *Endwert* alle vom Typ Integer sind. Achten Sie dabei besonders auf die Fälle *Anfangswert* = *Endwert* beziehungsweise *Anfangswert* > *Endwert*.

```
FOR i := Anfangswert TO Endwert DO
WriteLn (i);
```

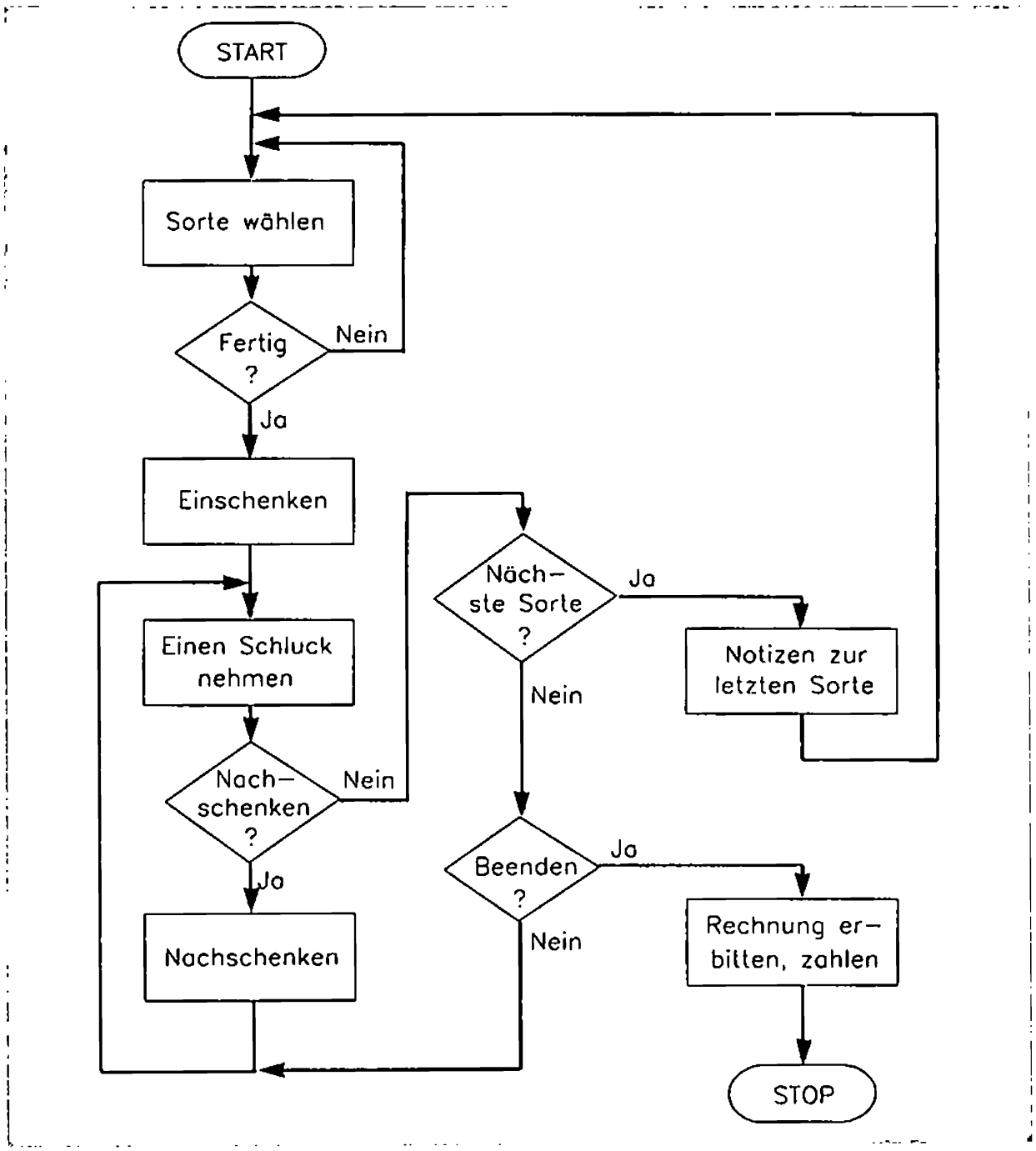

Abbildung 9-8: Setzen Sie das Struktogramm in Pascal um!

10 Textdateien und STRING-Verarbeitung

Dateien in Pascal – Abarbeiten einer Textdatei – Daten speichern und wieder holen – Text Generators Ltd. – STRING-Spezialitäten – Silbentrennprogramm – Aufgaben

Ein Programm ist nicht nur in der Lage, Tastatureingaben und Bildschirmausgaben zu steuern. Es kann Textströme auch auf Diskette, Platte oder andere Permanentspeicher lenken und dort wieder herholen. In diesem Kapitel „lernen" Pascal-Programme das Lesen und Schreiben auf Disketten bzw. Platte. Die Wiederholungsstruktur wird hier ihre typischste Anwendung finden: Ein beliebig umfangreicher Datenvorrat kann unermüdlich nach einem vorgegebenen Muster abgearbeitet werden.

„Quellen" und „Senken" von Zeichenströmen

In Kapitel 2 wurde der Computer als ein universelles Gerät zur Verarbeitung von Zeichen definiert. Zugrunde lag die Vorstellung eines Stroms von Symbolen, die, vom Computer aufgenommen, in irgendeiner Weise transformiert und dann ausgegeben werden. Das damalige Schema wird nun etwas weiter ausgeführt, indem auch die möglichen Herkunfts- und Zielorte für den Zeichenstrom betrachtet werden. In der abstrakten Theorie „sprudeln" Zeichen aus Quellen und „sammeln sich" in Senken.

Die Quelle beziehungsweise die Senke für einen Zeichenstrom auf dem Computer ist entweder ein Gerät (wie Bildschirm oder Tastatur; englisch „device") oder eine Datei. In Pascal werden Geräte als Spezialfälle von Dateien behandelt.

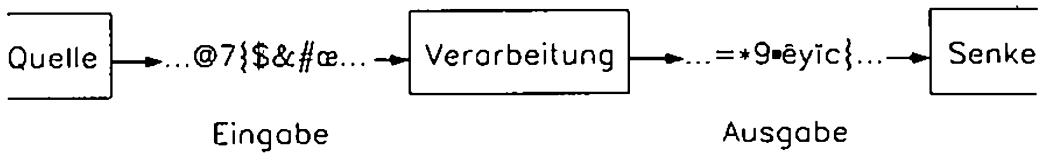

Abbildung 10-1: Erweitertes Computerschema

In diesem Sinn wird die Tastatur als Datei, als Quelle eines Zeichenstroms aufgefaßt. Von ihr kommt potentiell eine unaufhörliche Folge von Zeichen. Genauso zählen das Bildschirmgerät und der Drucker als Dateien, in die unbegrenzt Zeichen transportiert werden können. Diskettendateien sind dagegen nicht unerschöpflich. Aber ihre Speicherkapazität ist ziemlich groß, und sie bewahren den Zeichenstrom für die spätere Verwendung auf. Mit ihnen haben Sie bislang gearbeitet, wenn Sie Programm- oder andere Editortexte gespeichert und geladen haben. Was hier das Editorsystem besorgt, das Schreiben und Lesen auf und von Diskette oder Platte, können Sie auch mit einfachen Pascal-Programmen selber veranlassen. Die Anweisungen, die Sie hierfür benötigen, sind im Grunde die vertrauten Write- und Read-Operationen, die lediglich anstatt mit der Standardsenke Bildschirm beziehungsweise der Standardquelle Tastatur mit anderen Dateien arbeiten müssen.

Dateivariable

Zur Bezeichnung von Dateien werden Dateivariable verwendet. Die jeweilige Dateivariable, etwa *Quelle* beziehungsweise *Senke*, wird entsprechend dem folgenden Schema als erstes Glied in die Klammer hinter Read oder ReadLn, Write oder WriteLn geschrieben:

```
Read    (Quelle {, ... } );
Write   (Senke  {, ... } );
ReadLn  (Quelle {, ... } );
WriteLn (Senke  {, ... } );
```

Nach der Angabe der Quelle oder der Senke folgen wie üblich die Variablen, die eingelesen, oder die Ausdrücke, die geschrieben werden sollen. Wenn lediglich ein Return geholt beziehungsweise ein Wagenrücklaufzeichen abgeschickt werden soll, stehen die Dateivariablen allein in der Klammer:

```
ReadLn  (Quelle);
WriteLn (Senke);
```

Vordefinierte Dateivariable

Das Prinzip haben Sie schon am Beispiel der Ausgabe auf dem Drucker kennengelernt, für den es in Turbo-Pascal die vordefinierte Dateivariable *LST* gibt (ab Turbo 4.0 zugänglich, wenn in der USES-Deklaration der Modul *Printer* aufgezählt wird). Sofern Sie mit Turbo 3.x arbei-

ten, haben Sie mit dem ungepufferten und echofreien Eingabekanal KBD auch schon eine vordefinierte Quelldatei gesehen.

Deklaration von Text-Variablen

Werden weder die Standarddateien noch andere vordefinierte Dateivariable benutzt, so sind die Variablen wie andere Variable zu deklarieren. Für die formal unstrukturierten Textdateien gibt es den speziellen Typ *Text*. Damit die oben aufgeführten Beispiele funktionieren, muß also (unter der Überschrift VAR) deklariert werden:

```
Quelle, Senke: Text;
```

Die Brücke zwischen Pascal-Programm und Betriebssystem

Dateien sind für ein Programmiersystem eine etwas heikle Angelegenheit, weil die externen Ein- und Ausgabegeräte (Diskettenlaufwerke, Tastatur, Bildschirm und so weiter) bedient werden müssen. Die aufwendige Verwaltung der Verbindungen zu diesen Geräten ist grundsätzlich Sache des Betriebssystems. Ein Pascal-Programm, das auf diverse Dateien zurückgreifen soll, kann sich daher nicht auf seine innere Logik beschränken, sondern muß in besonderem Maß an die Mittel des Betriebssystems anknüpfen. Eine Diskettendatei trägt zum Beispiel einen nach den Vorschriften des Betriebssystems gebildeten Namen. Sie kann unter diesem Namen aufgerufen und abgespeichert werden. Der Aufruf und die Organisation der Ablage sind Obliegenheiten des Betriebssystems. Das Pascal-Programm benutzt dagegen eine nach Pascal-Regeln definierte Dateivariable und soll mit ihr arbeiten können. Daher muß eine Brücke zwischen dem Betriebssystem und den internen Bezeichnungen und Ablaufregeln des Pascal-Programms geschlagen werden. Die Betriebssystemaufrufe müssen in die Pascal-Umgebung übersetzt werden.

Unterschiede in der Dateibehandlung bei verschiedenen Pascal-Implementationen

Leider ist diese Übersetzung je nach Pascal-Dialekt etwas unterschiedlich ausgefallen. An der Einbruchsstelle des Betriebssystems finden sich relativ große Abweichungen zwischen den verschiedenen Pascal-Implementationen (so wird die Einrichtung eines speziellen Pascal-Systems genannt). Sollten Sie ein anderes System als Turbo- oder Quick-Pascal verwenden, müssen Sie auf jeden Fall in Ihrem Handbuch nachsehen, wie die Programmbeispiele abzuändern sind.

Assign

Für den ersten Brückenschlag zwischen Betriebssystem und den pascalinternen Größen gibt es in Turbo-Pascal eine eigene Anweisung, *Assign* („zuordnen"). Mit Assign weisen Sie der im Programm deklarierten Pascal-Dateivariablen den Namen einer Betriebssystemdatei zu. Der Dateiname kann als fester STRING (in Apostrophen) oder mittels einer STRING-Variablen bezeichnet werden. Zum Beispiel:

```
Assign (Quelle, 'A:Brief.txt');
Assign (Senke,  'B:Adresse.txt');
```

Oder, wenn *Dateiname1, Dateiname2* als STRING deklarierte Variablen sind:

```
Assign (Quelle, Dateiname1);
Assign (Senke,  Dateiname2);
```

Dateien öffnen und schließen

Das Betriebssystem reserviert für jede Datei, auf die Lese- und Schreibzugriffe stattfinden sollen, einen gewissen Bereich des internen Speichers als Puffer. Dadurch muß nicht bei jedem zu lesenden oder zu schreibenden Zeichen auf das langsame Disk-Laufwerk zugegriffen werden. Es wird jeweils ein Block von Zeichen in einem Rutsch geholt beziehungsweise losgeschickt, der im Puffer zwischengespeichert wird. Damit nicht zu viel Speicherplatz für diese Puffer verloren geht, erlaubt das Betriebssystem nur eine beschränkte Anzahl gleichzeitig offener Dateien. Eine Datei muß daher zu ihrer Benutzung ausdrücklich eröffnet werden. Wenn sie nicht gebraucht wird, sollte sie wieder geschlossen werden, um den Dateipuffer freizugeben.

Öffnen mit ReWrite

Je nach Art der beabsichtigten Dateinutzung gibt es zwei Anweisungen für das Öffnen:

```
ReWrite (Senke);
```

ReWrite richtet die Betriebssystemdatei, die in der Assign-Anweisung mit *Senke* verbunden wurde, zum Schreiben neu ein. Handelt es sich um eine Disk-Datei, wird sie neu angelegt – eine eventuell schon vorhandene Datei gleichen Namens wird überschrieben.

Öffnen mit Reset

Wollen Sie dagegen eine vorhandene Datei von vorne anfangend lesen, so öffnen Sie diese mit Reset:

```
Reset (Quelle);
```

Es gibt zwar Spezialbefehle, doch mit den Standardanweisungen kann zu einer bestehenden Datei nichts hinzugefügt werden. Entweder wird eine vorhandene Datei gelesen oder eine neue erstellt. Jede Veränderung oder Erweiterung der Datei erfordert, daß eine neue Version von ihr auf der Diskette erzeugt wird. Gelesen wird eine Textdatei stets von vorne, ohne daß Abschnitte übersprungen werden. Immerhin dürfen Sie mit einem einfachen *Reset (Quelle)* an den Anfang der Datei zurückkehren, um sie erneut zu lesen.

Close

Die Anweisung zum Schließen der Dateien verwendet einfach das Wort *Close*:

```
Close (Quelle);
Close (Senke);
```

Wenn Sie eine Datei geschlossen haben, ist die Dateivariable wieder frei. Mit einem neuen Assign können Sie ihr eine andere Disk-Datei zuweisen. Es sei nochmal darauf hingewiesen, daß Sie, wenn Sie ein anderes System als Turbo- oder Quick-Pascal benutzen, genau in Ihrem Handbuch nachschauen sollten, wie Textdateien behandelt werden. Es gibt Pascal-Dialekte, bei denen eine Datei nur dauerhaft auf Diskette bleibt (und nicht automatisch gelöscht wird), wenn bei *Close* der Zusatz *lock* („verschließen") angegeben ist.

Abarbeiten einer Textdatei

EOF

Das Betriebssystem teilt dem Programm auch mit, ob beim Lesen das Dateiende erreicht wurde. In Pascal kann diese Information abgefragt werden mit der Funktion *eof* (eine Abkürzung für End Of File – „Ende der Datei"):

```
eof (Quelle)
```

eof liefert einen booleschen Wert.

Eine Datei insgesamt lesen heißt: „Solange noch Daten in der Disk-Datei sind, lies und verarbeite sie". Benötigt wird also eine Wiederholungsstruktur, deren Finalbedingung mit Hilfe eines eof-Ausdrucks formuliert werden kann. Wenn als Quelle *Text* deklariert wurde, *Zeile* und *Dateiname* STRING-Variablen sind und nach einer Eingabe oder einer Wertzuweisung der Name einer Datei auf *Dateiname* steht, so lautet der typische Ablauf eines Pas-

cal-Programms zur Verarbeitung einer Textdatei wie folgt:

```
Assign (Quelle, Dateiname);
Reset  (Quelle);
WHILE NOT eof (Quelle) DO
  BEGIN
    ReadLn  (Quelle, Zeile);
    WriteLn (Lst, Zeile);
  END;
Close (Quelle);
```

Datei mit Zeilenstruktur

Im diesem Fall wurde zugrunde gelegt, daß die Datei eine Zeilenstruktur besitzt, also zum Beispiel mit einem Editor erzeugt wurde. In der Textdatei sind die Return-Zeichen, die am Zeilenende erzeugt wurden, mit abgespeichert. Beim Lesen der Datei werden sie durch ReadLn so behandelt, als kämen sie unmittelbar von der Tastatur. Jedes ReadLn nimmt daher den Text einer ganzen Zeile auf und überträgt ihn in die Variable *Zeile*.

In den Anweisungen oben wird *Zeile* zum Drucker ausgegeben. Somit ergibt sich ein einfaches Ausdruckprogramm, das Sie später mit mehr Komfort ausstatten können.

Zeichenweises Lesen

Wenn die Datei keine Zeilenstruktur besitzt, kann man sie auch zeichenweise einlesen. Anstelle einer STRING-Variablen wird eine Char-Größe, etwa *Zeichen*, und Read statt ReadLn verwendet. Ansonsten bleibt der Ablauf unverändert. Das folgende Beispiel kann dazu dienen, im Editor nicht lesbare Dateien, die vielleicht von anderen Textverarbeitungssystemen erzeugt wurden, ein bißchen zu analysieren:

```
Assign (Quelle, Dateiname);
Reset  (Quelle);
WHILE NOT eof (Quelle) DO
  BEGIN
    Read (Quelle, Zeichen);
    Write (Zeichen, '  ', ord (Zeichen));
                          {Gibt das Zeichen und seine
                          ASCII-Codenummer aus.}
    ReadLn;               {Nächstes Zeichen mit Return}
  END;
Close (Quelle);
```

Daten speichern und wieder holen

Das folgende Turbo-Programm nimmt jeweils zwei (mit einem Würfel von Hand bestimmte) Werte von der Tastatur auf und bildet die Summe. Aber es schreibt das Ergebnis der Addition nicht auf den Bildschirm, sondern zeilenweise in die Datei *HISTO.TXT* im Laufwerk *B:* Das Programm WuerfelSpiel zeichnet sozusagen die Geschichte („Historie") eines Spiels auf. Die Aufzeichnung endet durch Eingabe eines Wertes, der zu groß als Würfelzahl ist.

```
PROGRAM WuerfelSpiel;

USES crt, printer; {Muß erst ab den Versionen Turbo-Pascal 4.0
                    eingefügt werden.}

VAR
  Geschichtsdatei      : Text;
  Summe, Wert1, Wert2  : Integer;
  MEHR                 : Boolean;
  Zeichen              : Chor;

PROCEDURE ProgrammErklaeren;
  BEGIN
    ClrScr;
    WriteLn;
    WriteLn ('Dieses Programm speichert ein "Würfelspiel" ');
    WriteLn ('in einer Datei A:\HISTO.TXT ab.');
    Write ('————————————————————');
    Write ('————————————————————');
    WriteLn;
    WriteLn ('Wollen Sie ein Würfelspiel abspeichern?');
    WriteLn; WriteLn;
    WriteLn ('Drücken Sie die Taste [ j ] für "Ja"!');
    WriteLn ('Jede andere Taste = Abbruch!');
    WriteLn; WriteLn;
  END;

PROCEDURE Speichern;
BEGIN
  Assign  (Geschichtsdatei,'A:Histo.txt');
  ReWrite (Geschichtsdatei);
  MEHR := true;
  WHILE MEHR DO
   BEGIN                    {Eingeben von Tastatur}
```

```
    ClrScr;
    GoToXY (1,15);
    Write  ('Beenden mit einer Zahl größer als 6!');
    GoToXY (1,5);
    Write  ('Was bringt der erste Wurf: ');
    ReadLn (Wert1);
    GoToXY (1,6);
    Write  ('Was der nächste          : ');
    ReadLn (Wert2);       {Summieren und Schreiben in Datei}
    Summe := Wert1 + Wert2;
    IF (Wert1 IN [1..6] AND (Wert2 IN [1..6])
      THEN
        WriteLn (Geschichtsdatei, Summe)
      ELSE
        MEHR :=false
  END;
  Close (Geschichtsdatei);
END;

BEGIN (* von Hauptprogramm *)
  ProgrammErklaeren;
  Zeichen : = ReadKey;
              {Turbo-Pascal 3.0: Read (Kbd, Zeichen}
              if Zeichen in ['j', 'J'] then Speichern;
END.
```

Man kann sich die gespeicherte Datei HISTO.TXT genauso wie jede andere Textdatei im Editor ansehen. Will man die Daten in irgendeiner Form automatisch verarbeiten, können sie auch mit einem eigenen Programm gelesen werden. Die Anweisungen lauten fast genauso, als würden die Daten nochmal mittels Tastatur eingegeben.

Ermittlung des Durchschnittswertes

Beispiel: um die durchschnittlich gewürfelte Augenzahl (den Mittelwert) zu bestimmen, holt man nacheinander die einzelnen Werte aus der Datei, summiert sie (mit Hilfe der Summenvariable *SummeAllerWerte*) auf und zählt mit, wieviele es sind. Nach der Schleife muß *SummeAllerWerte* durch *AnzahlWerte* dividiert werden. Das Programm sieht also folgendermaßen aus:

```
PROGRAM Auswertung;

USES crt, printer;

VAR
  Aufzeichnungsdatei : Text;
  Durchschnitt       : Real;
  Einzelwert, AnzahlWerte, SummeAllerWerte : Integer;

PROCEDURE ProgrammErklaeren;
  BEGIN
    ClrScr;
    WriteLn;
    WriteLn ('Diese Programm errechnet den Mittelwert');
    WriteLn ('für die in A:\HISTO.TXT gespeicherten.');
    WriteLn ('(Würfel)Zahlen.');
    Write ('————————————————————');
    Write ('————————————————————');
    WriteLn;
  END;

BEGIN
  ProgrammErklaeren;
  Assign (Aufzeichnungsdatei,'A:Histo.txt');
  Reset  (Aufzeichnungsdatei);
  AnzahlWerte      := 0;
  SummeAllerWerte  := 0;
  WHILE NOT eof (Aufzeichnungsdatei) DO
    BEGIN
      ReadLn (Aufzeichnungsdatei, Einzelwert);
      AnzahlWerte     := AnzahlWerte + 1;
      SummeAllerWerte := SummeAllerWerte + Einzelwert;
    END;
  Durchschnitt := SummeAllerWerte / AnzahlWerte;
  WriteLn ('Der Mittelwert beträgt: ', Durchschnitt :4:1 );
  Close (Aufzeichnungsdatei);
END.
```

Geräte als Dateien

Wenn Sie, um ein Programm zu testen, die Ausgabe kurzfristig auf den Bildschirm umlenken oder Eingaben zwischendurch von der Tastatur statt von einer Diskettendatei nehmen wollen, brauchen Sie die Dateivariable nicht aus dem Programmtext zu entfernen: Sie können in der Assign-Anweisung mit der Dateivariablen auch eine Gerätebezeichnung verbinden, das heißt einen Standard-

Dateinamen von Turbo-Pascal beziehungsweise MS-DOS für das entsprechende Ein- oder Ausgabegerät. Mit *CON* (console) kann die Tastatur zur Eingabe und der Bildschirm zur Ausgabe angewählt werden. Zum Beispiel würden nach den beiden folgenden Zuordnungen die WriteLn- und ReadLn-Anweisungen aus dem obigen Programm so arbeiten, als wäre keine Dateivariable angegeben:

```
Assign (Geschichtsdatei,     'CON');
Assign (Aufzeichnungsdatei, 'CON');
```

Als weitere Standard-Gerätenamen können Sie *PRN* (printer) oder *LPT1* (für Line printer 1) für den ersten Drucker verwenden, *LPT2* und *LPT3* für einen zweiten und dritten angeschlossenen Drucker (in Turbo 3.x heißt es *LST*). Die seriellen Schnittstellen werden über *COM1* (oder auch *AUX*) und *COM2* angesprochen. Serielle Datenübertragung bedeutet, daß die einzelnen Bits eines Bytes nacheinander, nicht gleichzeitig bzw. parallel gesendet werden.

Die Bezeichnungen der Geräteeinheiten sind spezifisch für das Betriebssystem. Außerdem können die Entwickler der jeweiligen Pascal-Version noch eigene Erfindungen hinzutun oder vom Betriebssystem angebotene sperren. Sogar die Abweichungen zwischen den verschiedenen Versionen von Turbo-Pascal sind an diesem Punkt erheblich, die Gründe für die diversen Veränderungen nicht immer nachvollziehbar. Ein Programm, das solche Geräteeinheiten benutzt, muß beim Wechsel zu einer anderen Pascal-Implementation überprüft und in der Regel abgeändert werden.

Text Generators Ltd.

Obwohl es für fast alle Zwecke fertige Programme zu kaufen gibt, kann es nützlich sein, kleinere Datenanpassungen mit eigenen, relativ schnell geschriebenen Programmen selbst durchzuführen. Eröffnen wir also eine kleine Firma für Programme zur automatischen Textbearbeitung: *Text Generators Limited*. Wie stets beim Computer handelt es sich hierbei um ein Hilfsmittel zur formalen Bearbeitung von Zeichenströmen.

Fließtext erzeugen

Ein Editor arbeitet zeilenorientiert, das heißt, jede Bildschirmzeile wird direkt abgespeichert, indem das Zeilen-

ende mit der Return-Zeichenkombination (Wagenrücklauf/ Zeilenvorschub) markiert wird. Textverarbeitungssysteme benötigen in der Regel Fließtext, Return steht hierbei nur am Ende eines Absatzes. Sie können im Editor geschriebene Texte (bei denen die Absätze durch Leerzeilen angezeigt sind) automatisch in die andere, absatzorientierte Fließtextform überführen, indem Sie den Text zeilenweise mit ReadLn lesen und die gelesenen STRINGs nur mittels Write, also ohne Return-Zeichen, in eine neue Datei schreiben. WriteLn wird nur aufgerufen, wenn die gelesene Zeile leer war:

```
PROGRAM FliessTextErzeugen;

USES crt, printer; {Muß erst ab den Versionen Turbo-Pascal 4.0
                    eingefügt werden.}

VAR
  Quelle, Senke            : Text;
  QuellenName, SenkenName  : STRING [40];
  Zeile                    : STRING [255];

PROCEDURE DateienBestimmen;
  BEGIN
    Write  (' umzuwandelnde Datei: ');
    ReadLn (QuellenName);
    Write  (' neu erzeugte Datei:  ');
    ReadLn (SenkenName);
  END;

PROCEDURE ProgrammErklaeren;
  BEGIN
    ClrScr;
    WriteLn;
    Write ('Das Programm wandelt einen zeilenorientiert');
    Write ('gespeicherten Text in Fließtext um');
    WriteLn;
    Write ('————————————————————');
    Write ('————————————————————');
    WriteLn;
    WriteLn;
  END;

BEGIN (* von Hauptprogramm *)
  ProgrammErklaeren;
  DateienBestimmen;
  Assign (Quelle, QuellenName);
  Assign (Senke, SenkenName);
```

```
  Reset (Quelle);
  ReWrite (Senke);
  WHILE NOT eof (Quelle)
    DO
      BEGIN
        ReadLn (Quelle, Zeile);
        Write (Senke, Zeile);
        IF Zeile = ''
          THEN
            BEGIN
              WriteLn (Senke);
              WriteLn (Senke);
            END    {Zweimal Absatz entspricht}
      END;         {Absatzende plus Leerzeile.}
  Close (Quelle);
  Close (Senke);
END.
```

Wieder Zeilen erzeugen

Den entstandenen Text können Sie mit vielen Editoren nicht mehr lesen, weil diese nur begrenzte Zeilenlängen verarbeiten. Wenn Sie den Text wieder zurück in eine Datei mit Zeilenstruktur wandeln wollen, können Sie ihn zeichenweise aufnehmen und die Zeichen mitzählen. Sobald eine Zeile ungefähr voll ist, schreiben Sie mittels WriteLn ein Return in die neue Datei:

```
PROGRAM ZeilenstrukturErzeugen;

USES crt, printer; {Muß erst ab den Versionen Turbo-Pascal 4.0
                    eingefügt werden.}

CONST
  MindestZeilenlaenge = 55;
VAR
  Quelle, Senke            : Text;
  QuellenName, SenkenName  : STRING [40];
  Zeichen                  : Char;
  ZeichenInEinerZeile      : Integer;

PROCEDURE DateienBestimmen;
  BEGIN
                           {Wie bisher}
  END;
```

```
PROCEDURE ProgrammErklaeren;
  BEGIN
              {Wie bisher; allerdings muß die Texterklärung
              gemäß der Aufgabe des Programms abgeändert werden.}
  END;

BEGIN (* von Hauptprogramm *)
  ProgrammErklaeren;
  DateienBestimmen;
  Assign (Quelle, QuellenName);
  Assign (Senke, SenkenName);
  Reset (Quelle);
  ReWrite (Senke);
  ZeichenInEinerZeile := 0;
  WHILE NOT eof (Quelle) DO
    BEGIN
     Read (Quelle, Zeichen);
     ZeichenInEinerZeile := ZeichenInEinerZeile + 1;
     IF (ZeichenInEinerZeile > MindestZeilenlaenge)
              {Zeile bald voll}
     AND (Zeichen = ' ')
              {Das gelesene Zeichen ist das Leerzeichen,
              das heißt, ein Wort ist zu Ende.}
       THEN
              {Sie schreiben ein Zeilenende und setzen den
              Zähler auf 0}
         BEGIN
           WriteLn (Senke);
           ZeichenInEinerZeile := 0;
         END
       ELSE
         BEGIN
              {Das gelesene Zeichen wird in die neue Datei
              geschrieben.}
           Write (Senke, Zeichen);
              {Wenn das Zeichen "Wagenrücklauf" ist, muß
              der Zähler auch auf 0 gesetzt werden:}
           IF Zeichen = chr (13)
              THEN ZeichenInEinerZeile := 0;
         END;
    END;
  Close (Quelle);
  Close (Senke);
END.
```

EoLn

In manchen Pascal-Dialekten wird das Return-Zeichen von einem einfachen Read nicht aufgenommen. Bei obigem Programm würden daher die ursprünglich im Text vorhandenen Zeilenendezeichen nicht in die neue Datei übertragen. Die ursprünglichen Zeilenschlüsse, das heißt die Absätze, würden verschwinden – was nicht beabsichtigt ist. Auch die Abfrage *Zeichen = chr (13)* würde nichts bewirken. Um ein Zeilenende festzustellen, muß dann folgende (auch in Turbo-Pascal vorhandene) boolesche Funktion verwendet werden:

```
EoLn (Quelle)
```

Sie ergibt den Wert true, wenn die Zeile zu Ende ist, sonst false (EoLn bedeutet End of Line). Die Bedingung für WriteLn (Senke) müßte ergänzt werden um:

```
OR EoLn (Quelle)
```

Dafür läßt sich die IF-Anweisung im ELSE-Teil der großen Fallunterscheidung einsparen.

Das Programm *ZeilenstrukturErzeugen* kann zu einer halbautomatischen, interaktiven Silbentrennung erweitert werden. Das heißt, ein Benutzer oder eine Benutzerin soll Gelegenheit erhalten, Trennstellen für Wörter zu nennen, die in das Stück zwischen *MindestZeilenlaenge* und einer noch festzulegenden *Maximallaenge* nicht passen. Damit dies mit einem gewissen Komfort geschieht, müssen solche Textstücke (des Datentyps STRING) zwischengespeichert und bei Bedarf auseinandergenommen, gekürzt und ergänzt werden können. Für derartige Verarbeitungsaufgaben gibt es in den meisten Pascal-Varianten eine Reihe von fertigen Funktionen und Prozeduren.

STRING-Spezialitäten

Length und Pos

Wenn *Wort* eine STRING-Größe ist, dann teilt *Length (Wort)* die Anzahl der Zeichen mit, die *Wort* aktuell umfaßt (die „Länge"). Es handelt sich natürlich um einen Integer-Wert. Eine äußerst praktische Funktion ist auch *Pos* (Position):

```
Pos (Stueck, Wort)
```

Die Position von *Stueck* in *Wort*, besagt, beim wievielten Zeichen von *Wort* der STRING *Stueck* beginnt (bei mehrfachem Vorkommen zählt das erste). Offenbar er-

gibt sich wieder ein Integer-Wert. Zum Beispiel hat *Pos ('ch','Buch')* den Wert 3. Sofern *Stueck* keinen Teil von *Wort* bildet, bekommt *Pos (Stueck, Wort)* den Wert 0.

Dies führt auf eine einfache Methode, um festzustellen, ob ein bestimmter STRING in einer Zeile enthalten ist. Soll das erste Beispiel dieses Kapitels so abgeändert werden, daß automatisch am Anfang einer Prozedurdeklaration im Programmausdruck eine Leerzeile eingeschoben wird, fragen Sie ab:

```
IF Pos ('PROCEDURE', Zeile) <> 0
```

Wenn dies der Fall ist, lassen Sie vor der Ausgabe von *Zeile* ein WriteLn (Senke) ausführen. Das Verfahren funktioniert in dieser Form nur, wenn PROCEDURE immer exakt gleich geschrieben wurde.

Concat

Um zwei oder mehrere Strings aneinanderzuhängen, existiert die Funktion *Concat*, (concatenation, „Verkettung"). Das Resultat kann einer STRING-Variablen zugewiesen oder anderweitig verwendet werden. Wenn *Anrede* und *Zeile* STRING-Variablen sind (wobei auch mehrere STRING-Variablen vorkommen dürfen oder dieselben mehrfach), kann folgender Ausdruck gebildet werden:

```
Zeile :=  Concat ('Liebe', Anrede, '!');
```

In Turbo-Pascal lassen sich Textstücke nicht nur mit Concat verketten, sondern auch mit einem Pluszeichen „addieren".

```
Zeile :=  'Liebe' + Anrede + '!';
```

Mit dieser Zuweisung sind die folgenden Anweisungen gleichbedeutend:

```
WriteLn (Zeile); {Dasselbe wie:}
WriteLn ('Liebe', Anrede, '!');
```

Verketten von Chars und Strings

Ist das Ausrufezeichen im Beispiel eigentlich ein STRING oder ein Char? Sie müssen sich zum Glück nicht entscheiden, weil STRING-Werte der Länge 1 wie Chars behandelt werden. Kritischer wird es, wenn statt eines Werts eine Char-Variable, etwa *SchlussZeichen* verwendet werden muß. Strenggenommen lassen sich mit Concat beziehungsweise + nur Strings verknüpfen. In Turbo-Pascal dürfte dennoch gebildet werden:

```
Zeile :=  'Liebe' + Anrede
          + SchlussZeichen;
```

Andere Pascal-Dialekte sehen das enger. Dann muß ein Trick weiterhelfen. Erst wird irgendein festes Zeichen an den STRING gebunden (zum Beispiel das Leerzeichen) und dann das Zeichen durch den Inhalt der Variablen ersetzt. Das sieht etwa so aus:

```
Zeile := Concat ('Liebe', Anrede, ' ');
Zeile [Length (Zeile)]  := SchlussZeichen;
```

Wenn hinter einer STRING-Variablen in eckigen Klammern eine Nummer steht (etwa 15), bedeutet dieser Ausdruck das entsprechende Zeichen (im Beispiel das fünfzehnte) des STRINGs. *Length (Zeile)* gibt die Länge und damit die Nummer des letzten Zeichens von *Zeile* an. Mit *Zeile [Length (Zeile)]* können Sie also das letzte Zeichen von *Zeile* ansprechen.

Copy

Mittels der Funktion *Copy* kann ein Ausschnitt eines STRINGs *Wort* auf eine andere STRING-Variable *Teil* kopiert werden. Nach Ausführung der Anweisung:

```
Teil := Copy (Wort, i, z)
```

enthält *Teil* die z Zeichen, die in *Wort* ab dessen i-ten Zeichen stehen. Zum Beispiel ergibt die folgende Programmzeile den STRING *'uch'*:

```
Copy ('Buch', 2, 3)
```

Funktionen und Prozeduren

Was haben die bisher geschilderten STRING-Spezialitäten gemeinsam? Sie tauchen als Teil einer Wertzuweisung oder als Teile anderer Anweisungen auf, wo sie für *Werte* stehen. Da mathematische Funktionen in ähnlicher Form Werte repräsentieren, wird von STRING-*Funktionen* gesprochen. Wenn dagegen eine STRING-*Prozedur* aufgerufen wird, so stellt sie für sich, wie jeder anderer Prozeduraufruf, eine komplette Anweisung dar.

Delete und Insert

Unter Verwendung zweier Prozeduren können gezielt Teile eines Strings herausgeschnitten oder Stücke eingefügt werden. Das erste sieht so aus:

```
Delete (Wort, i, z);
```

Delete löscht, beginnend an der Stelle *i*, z Zeichen in der STRING-Variablen *Wort*. Ein Beispiel verdeutlicht die Arbeitsweise:

```
Wort := 'Buchstabe';
Delete  (Wort, 3, 5);
WriteLn (Wort);
```

Das dritte Zeichen ist das „c", fünf abgezählt führt auf „a": Am Ende bleibt der „Bu..be" übrig. Das Gegenstück zu Delete ist die Prozedur *Insert*, mit der etwas in einen STRING eingebaut werden kann:

```
Insert (Stueck, Wort, i);
```

Hier wird *Stueck* an der i-ten Stelle von *Wort* in diesen STRING eingefügt. Die nachfolgende Zeile hebt obige Löschung wieder auf.

```
Insert ('chsta', 'Bube', 3);
```

Jetzt haben Sie viele Möglichkeiten, STRINGs zu verketten, zu verschieben, zu kürzen, zu ergänzen. Was nützt Ihnen das konkret, wenn es darum geht, die Ausgabe von errechneten Werten zu gestalten? Auch dafür ist vorgesorgt. Wenn für ein Programm die Standardformatierungen für Zahlen nicht genügen, können Sie mittels der Prozedur *Str* eine Zahl in einen STRING umwandeln, der dann weiter bearbeitet wird: **Str**

```
Str (Zahl, Wort)
```

Diese Programmzeile schreibt auf den STRING *Wort* die Zeichenfolge, die dem numerischen Wert *Zahl* entspricht. Turbo-Pascal kennt eine Prozedur, die das Umgekehrte macht: *Val (Wort, Zahl, Pruefwert)* versucht eine Zahlenbedeutung zu ermitteln, die der auf *Wort* stehende STRING hat, und setzt das Ergebnis auf die Real- oder Integer-Variable *Zahl*. Natürlich lassen sich sinnvoll nur STRINGs als Zahlen interpretieren, die aus Ziffern bestehen beziehungsweise ein gültiges Dezimalzahlenformat haben. Daher wird in der dritten (Integer-) Variablen, hier *Pruefwert* genannt, festgehalten, ob die Umwandlung stattfand. Wenn es nicht oder nicht vollständig funktioniert hat, steht in *Pruefwert* die Position des ersten Zeichens im STRING, das nicht in das Zahlenformat paßte. Allerdings haben die Turbo-Konstrukteure früher einen Grenzfall übersehen: Auch bei einem leeren STRING wurde *Pruefwert* auf 0 gesetzt, als wäre die Zahl korrekt. Ab Turbo 4.0 ist der Fehler behoben. *Pruefwert* erhält beim Versuch, einen leeren STRING in eine Zahl zu transformieren, nun den Wert 1. **Val**

Das Silbentrennprogramm

Jetzt sind die Mittel beisammen, um das interaktive Silbentrennprogramm zu schreiben. Die Kombination des fließtexterzeugenden mit dem erweiterten, zeilenbildenden Programm gestattet, im Editor geschriebene Texte in bester Form auszugeben. Auf diese Weise können Sie den gewohnten Editor zur Textverarbeitung verwenden (Erläuterungen zum Programm folgen unten):

```
PROGRAM ZeilenBilden;
USES crt, printer; {Muß erst ab den Versionen Turbo-Pascal 4.0
                    eingefügt werden.}

CONST
  MindestZeilenlaenge = 55;
  Maximallaenge       = 60;
  AnzeigezeileNr      = 25;
  HalbeLeerzeile  =  '
                         {40 Leerzeichen}
  Bindestrich     =  '-';
  Trennzeichen    =  '-'; {Wenn der erzeugte Text später weiter
                           bearbeitet werden soll, ist es sinnvoll,
                           vom Bindestrich, ASCII-Nummer 45, einen
                           Trennstrich, intern meist das
                           ASCII-Zeichen 31, zu unterscheiden.}

VAR
  Quelle, Senke            : Text;
  QuellenName, SenkenName  : STRING [40];
  Zeichen                  : Char;
  Zeile, Restzeile         : STRING [255];

PROCEDURE ProgrammVorstellung;
  BEGIN
    ClrScr;
    WriteLn ('            Interaktive Silbentrennung');
                          {Mit 25 Leerzeichen eingeben.}
    GoToXY (20,3);
    Write  ('————————————————————');
  END;

PROCEDURE DateienBestimmen;
  BEGIN
    GoToXY (1, 6);
    Write  ('Ausgangsdatei: ');
    ReadLn (Quellenname);
    Write  ('Zieldatei:     ');
```

```
    ReadLn (SenkenName);
  END;

PROCEDURE Bedienungshinweis;
  BEGIN
    GoToXY  (1, 2);
    Write   (HalbeLeerzeile); Write (HalbeLeerzeile);
    GoToXY  (1, 3);
    Write   (HalbeLeerzeile); Write (HalbeLeerzeile);
    GoToXY  (1, 3);
    Write
('═══════════════════════════════════════════════════════════');
                    {Balken mit ASCII-Code: 220}
    GoToXY  (1, 1);
    WriteLn ('Trennstelle nach links: <  Nach rechts: > ',
            '  Hier trennen: - ');
    GoToXY  (1, 2);
    Write   ('Nicht trennen: Leertaste (Space) drücken.');
    GoToXY  (1, 4);
    Write   (HalbeLeerzeile); Write (HalbeLeerzeile);
  END;

PROCEDURE InteraktiveTrennung;
  VAR
    i: Integer;
    Getrennt, WortendeGefunden: Boolean;
    Eingabe: Char;

  PROCEDURE VerarbeiteEingabe;
  BEGIN
  IF Eingabe = '<'
    THEN                    {Gehe rückwärts}
      BEGIN
        IF i > 0 THEN i:= i - 1;
        GoToXY (i, AnzeigezeileNr);
      END
    ELSE
      IF Eingabe = '>'
        THEN                {Gehe wieder vorwärts}
          BEGIN
            IF i < Maximallaenge THEN i:= i + 1;
            GoToXY (i, AnzeigezeileNr);
          END
        ELSE
          IF Eingabe = '-'
            THEN
              BEGIN
```

```
            Getrennt := true;
            Restzeile := copy (Zeile, i, Maximallaenge + 1 - i
                                          + 1);
            Zeile := copy (Zeile, 1, i-1);
            Zeile := Zeile + Trennzeichen;
          END
        ELSE
          IF Eingabe = ' '
            THEN
              BEGIN
                Getrennt := true;
                      {Suche Wortende }
                WortendeGefunden := false;
                i:= MindestZeilenlaenge;
                WHILE NOT WortendeGefunden AND (i > 1) DO
                  IF Zeile [i] IN [' ',Bindestrich]
                    THEN WortendeGefunden := true
                    ELSE i:= i-1;
                IF NOT WortendeGefunden
                  THEN
                      {Die ganze Zeile nehmen.}
                    BEGIN
                     Restzeile
                            := copy (Zeile, Maximallaenge+1,1);
                     Zeile  := copy (Zeile, 1, Maximallaenge);
                    END
                  ELSE
                      {Wortende wurde gefunden.}
                    BEGIN
                      Restzeile := copy (Zeile, i+1,
                                         Maximallaenge + 1 - i);
                      Zeile := copy (Zeile, 1, i);
                    END;
              END                    (* von IF Eingabe := ' ' *)
        ELSE;          {Keine Reaktion}
  END (* von VerarbeiteEingabe *);

BEGIN (* von InteraktiveTrennung *)
  i:= Maximallaenge;
  GoToXY (1, AnzeigezeileNr);
  Write (Zeile);          {Anzeige der gelesenen Zeile}
  Bedienungshinweis;
  GoToXY (i, AnzeigezeileNr);
  Getrennt := false;
  WHILE NOT Getrennt DO
```

```
    BEGIN
      Eingabe := ReadKey;
                        {In Turbo Pascal 3.x: Read (kbd,
                        Eingabe);}
      VerarbeiteEingabe;
    END;
END (* von InteraktiveTrennung *);

PROCEDURE Trennen; {In dieser Prozedur wird entweder zwischen
                    zwei Wörtern oder mittels Silbentrennung die
                    Zeile aufgetrennt. Der Rest kommt auf
                    Restzeile, der Anfang bleibt auf Zeile.}
VAR
  i: Integer;
  WortendeGefunden: Boolean;

BEGIN
      {Zunächst suche im Zeilenstück zwischen Mindest- und
      Maximallaenge nach Wortgrenzen, so daß Silbentrennung
      nur wenn erforderlich angewandt wird. Starte die
      Suche bei letztem zulässigen Zeichen.}
  i := Maximallaenge;
  WortendeGefunden := false;
  WHILE NOT WortendeGefunden AND (i > MindestzeilenLaenge) DO
    BEGIN
      IF Zeile [i] IN  [' ',Bindestrich]
        THEN
          BEGIN
            Restzeile := copy (Zeile, i+1, Maximallaenge + 1 - i);
            Zeile := copy (Zeile, 1, i);
            WortendeGefunden := true;
          END;
      i := i-1;
    END;
  IF NOT WortendeGefunden
    THEN
      InteraktiveTrennung;
END (* von Trennen *);

PROCEDURE ZeileBearbeiten;
VAR
  x: Integer;

BEGIN
  Zeichen := ' ';
  WHILE NOT Eof (Quelle) AND NOT (Zeichen = chr (13))
                            AND (length (Zeile) < Maximallaenge+1) DO
                            {Füllen von Zeile}
```

```
    BEGIN
      Read (Quelle, Zeichen);
      IF NOT (Zeichen = chr (13)) THEN Zeile := Zeile + Zeichen;
    END;

  IF NOT eof (Quelle) AND (Zeichen = chr (13))
    THEN Read (Quelle, Zeichen);

              {Auf das Wagenrücklaufzeichen muß in einer Textdatei
              noch das Zeilenvorschubzeichen chr (10) folgen, das
              nicht weiter benötigt wird. Es wird deshalb
              gelesen und ignoriert.}

  IF (length (Zeile) <= Maximallaenge) OR (Zeichen = ' ')

              {Wenn Datei oder Zeile in der Quelldatei zu Ende ist
              oder am Zeilenende gerade ein Wortende erreicht ist,
              muß nicht getrennt werden.}

    THEN
      Restzeile := ''
    ELSE
      Trennen;
  WriteLn (Senke, Zeile);
  GoToXY (1, AnzeigezeileNr);
  Write (Zeile); {Anzeige der gelesenen Zeile}
  FOR x:= length (Zeile) TO Maximallaenge + 1 DO write (' ');
                {Löschen der abgetrennten Restzeichen}
  GoToXY (1,25);
  WriteLn;      {In der letzten Schirmzeile bewirkt dies, daß der
                ganze Schirminhalt nach oben geschoben wird.}
  Zeile := Restzeile; {Restzeile wird zum Anfang der nächsten
                      Zeile}
END (* von ZeileBearbeiten *);

BEGIN (* von Hauptprogramm *)
  ProgrammVorstellung;
  DateienBestimmen;
  Assign (Quelle, QuellenName);
  Assign (Senke, SenkenName);
  Reset (Quelle);
  ReWrite (Senke);
  Zeile := '';
  WHILE NOT eof (Quelle) DO
    ZeileBearbeiten;
  Close (Quelle);
  Close (Senke);
END. (* von Hauptprogramm *)
```

Zur Erklärung

Das Hauptprogramm ruft die Prozedur *ZeileBearbeiten* auf. Zur Bearbeitung einer Zeile gehört, daß das *Trennen* geregelt werden muß. Unter bestimmten Bedingungen erfordert *Trennen* eine *InteraktiveTrennung*. Obwohl logisch also eine klare Hierarchie besteht, sind die Prozeduren nicht ineinander gesetzt. Formal stehen die drei Prozeduren gleichrangig im Programm. Das schadet nichts, weil vorher im Deklarationsteil aufgeführte Prozeduren von später deklarierten und gleichrangigen aufgerufen werden können. Hier ist dies sinnvoll, da sonst die Schachtelung der Prozeduren unübersichtlich würde.

Ab Turbo-Pascal 4.0 können Sie mittels der Window-Prozedur auch einen Teil des Bildschirms zum Textausgabefenster erklären. Es wäre dann nicht mehr erforderlich, daß die Bedienungshinweise immer neu in die oberen Bildzeilen geschrieben werden.

Die Zeilenendeabfrage gestaltet sich beim zeichenweisen Lesen der Datei etwas mühsam, weil das Zeilenende üblicherweise durch die beiden Zeichen Carriage return („Wagenrücklauf", ASCII-Code 13) und Line feed („Zeilenvorschub", ASCII-Code 10) signalisiert wird. Wie oben schon erwähnt, gibt es Pascal-Dialekte, bei denen Sie von einer Read-Operation nie den Zeilenende-Code erhalten. Wenn Sie wissen wollen, ob die Zeile zu Ende ist, müssen Sie die Funktion EoLn benutzen. Das Programm kann sich dadurch sogar vereinfachen.

Im nächsten Kapitel wird eine Anweisung eingeführt, welche in einer Situation, wo eine umfangreiche Fallunterscheidung ansteht (etwa wie oben in der Prozedur *VerarbeiteEingabe*) das Aufschreiben der Fälle erleichtern kann.

Aufgaben

1. Stellen Sie sich vor, daß Sie nur Zugang zu einem Drukker ohne die deutschen Sonderzeichen haben. Sie müssen deshalb ß und die Umlaute in Buchstabenkombinationen auflösen. Schreiben Sie ein entsprechendes Programm.

2. Entwerfen Sie ein Ausdruckprogramm, mit dem Sie Ihre Programmcodes übersichtlich ausgeben können: mit Zeilennumerierung, korrektem Seitenvorschub, eventuell mit Seitennumerierung. Oft kann es praktisch sein, zusätzliche Leerzeichen an den Beginn jeder Zeile zu setzen,

um vom Papierrand wegzurücken. Sie können auch entsprechend dem Pascal-Text zusätzliche Einrückungen vornehmen oder Leerzeilen schreiben lassen. Die Möglichkeiten sind fast unbegrenzt.

3. In der Abteilung Text Generators können Sie ein Programm aufnehmen, das in einem Text besonders gekennzeichnete Zahlen heraussucht, deren Summen bildet und diese in den Ausgabetext setzt. Mit diesem Programm können Sie Ihre Rechnungen, Mahnungen etc. automatisch an unterschiedliche Daten anpassen. Sie können als Kennzeichnung etwa das Summenzeichen 'Σ' (ein griechisches großes Sigma) verwenden. Jede Zahl, die mit diesem Zeichen beginnt, wird aufsummiert, das Ergebnis (im durch die Sternchen beschriebenen Format) an der Stelle angezeigt, an der 'Σ' ohne folgende Ziffern steht. Ihr Programm würde dann aus dem Textauszug Abbildung 10-2 den Textauszug Abbildung 10-3 produzieren.

4. Werten Sie die Ökosystem-Simulation des letzten Kapitels grafisch aus. Lassen Sie das eigentliche Simulationsprogramm die Daten noch zusätzlich auf Diskette schreiben. Experimentieren Sie mit Programmen, die die Daten lesen, durch Dividieren und Runden auf eine passende Größe bringen und in ein Koordinatensystem eintragen.

```
... habe ich bereits am 11.3., 13.4. und 10.5. Teilbeträge
von DMΣ325, DMΣ290 und DMΣ310, insgesamt also
DMΣ****
überwiesen. Ich betrachte daher Ihre Mahnung vom 1.6.
als gegenstandslos.
...
Mit freundlichen Grüßen
...
```

Abbildung 10-2: Textauszug mit Summenzeichen

```
... habe ich bereits am 11.3., 13.4. und 10.5. Teilbeträge
von DM 325, DM 290 und DM 310, insgesamt also
DM 925
überwiesen. Ich betrachte daher Ihre Mahnung vom 1.6.
als gegenstandslos.
...
Mit freundlichen Grüßen
...
```

Abbildung 10-3: Textauszug ohne Summenzeichen

11 Menü-Steuerung von Programmen und Mehrfachauswahl

Zur Gestaltung einer Menü-Oberfläche – Mehrfachauswahl mit IF-THEN-ELSE – Die Menü-Schleife – Die CASE-Anweisung – Einfache Eingabekontrollen und die REPEAT-Anweisung – Die CASE-Anweisung außerhalb von Menüs – Aufgaben

Was darf es sein, soll der Computer wie ein Wirt den Benutzer oder die Benutzerin fragen und gleichsam eine Speisekarte vorlegen. Der Gast braucht nur auf eine der angebotenen Speisen zu zeigen, und das Gewünschte wird serviert. Ein solcher Umgang „à la carte" mit dem Computer beziehungsweise des Programms mit seinen Anwendern gilt als nutzungsfreundlich. Weil „Speisekarte" auf englisch „menue" heißt, wird das Verfahren „Menüsteuerung" beziehungsweise „Menüführung" genannt.

Beim Programmieren stellen Sie mit Hilfe von „Menüs" eine (teilweise willkürliche) logische Gliederung des bearbeiteten Problems her, die sich mehrstufig im Menübaum (der hierarchisch gegliederten Abfolge von Menüs, Untermenüs, Unter-Untermenüs und so fort) spiegelt. Die Menüs bilden zugleich die äußere Form, in der das Programm Mitteilungen macht und Eingaben entgegennimmt, das heißt seine Benutzungsoberfläche.

Zur Gestaltung einer Menü-Oberfläche

Forderungen

Als grundsätzliche Forderungen an die Gestaltung von Menüs lassen sich stellen:

- Menütexte sollten genau, plastisch und allgemeinverständlich sein.
- Sich rasch überlebende Wendungen und „Gags" sind zu meiden, ebenso unnötige Computerfachausdrücke oder eingestreute englische Begriffe in einem deutschen Text.

- Die Menüpunkte sollten erläutert sein, aber die Zusatzinformationen müssen grafisch auf dem Bildschirm abgesetzt erscheinen (so daß sie leicht übersprungen werden können, wenn das Programm vertraut geworden ist).
- Die Bildschirmfläche sollte klar und übersichtlich gegliedert werden. Für ähnliche Funktionen (Überschrift, Menüpunkte, Erklärung, Eingabe) ist grundsätzlich der gleiche Platz und dieselbe Bedienungsweise in den diversen Menüs oder Untermenüs vorzusehen.
- Die grafischen Möglichkeiten, die ein System bietet (Linien, unterschiedliche Helligkeiten) sollten zur Gestaltung genutzt werden, aber es ist sparsam und vorsichtig mit ihnen umzugehen (sonst irritieren die Effekte nur).
- Bei Farbbildschirmen sollten Texte, die in einem Zug gelesen werden, nicht in mehreren Farben erscheinen (für das Auge bedeutet jede Umstellung auf eine neue Bildschirmfarbe eine zusätzliche Anstrengung).

Aber nicht nur die Formulierungen in Programmeldungen und Menüs, sondern auch deren Abfolge, der Aufbau des Programms sollte weitestmöglich natürlich erscheinen. Das Programm sollte grundsätzlich entlang den Gesichtspunkten konstruiert sein, die eine Person in der entsprechenden Situation auch ohne das Programm berücksichtigen würde. Was die Bedienung angeht, soll ein Programm sich auch an unbewußten Erwartungen und Gewohnheiten, an Assoziationen und Verhaltensweisen orientieren. Der Erfolg der Maus als Gerät für die Bewegung des Cursors am Schirm beruht auf diesem Prinzip.

Mehrfachauswahl mit IF-THEN-ELSE

Beispiel Als Beispiel soll ein Programm wahlweise eine Glückwunschkarte zum Geburtstag, eine Einladungskarte zur Geburtstagsfeier, eine Einladung zum Klassentreffen oder eine Gratulationskarte zur bestandenen Prüfung auf den Bildschirm (oder auf den Drucker) zeichnen. Offensichtlich werden die vier Prozeduren *GeburtstagsKarte*, *GeburtstagsEinladung*, *KlassentreffenEinladung*, *PruefungsGratulation* benötigt. Eine Prozedur *MenueAnzeige* dient dazu, die Möglichkeiten, die das Programm bietet, zu er-

läutern. Je nach der Eingabe auf eine STRING-Variable *Anlass* soll die Prozedur für die entsprechende Karte aufgerufen werden. In der einfachsten Form sähe dann das Programm *KartenLaden* folgendermaßen aus:

```
PROGRAM KartenLaden;

USES crt, printer;

VAR
  Anlass: STRING [80];
PROCEDURE GeburtstagsKarte;
  BEGIN
                              {  ...  }
  END (* von GeburtstagsKarte *);

PROCEDURE GeburtstagsEinladung;
  BEGIN
                              {  ...  }
  END (* von GeburtstagEinladung *);

PROCEDURE KlassentreffenEinladung;
  BEGIN
                              {  ...  }
  END (* von KlassentreffenEinladung *);

PROCEDURE PruefungsGratulation;
  BEGIN
                              {  ...  }
  END (* von Pruefungsgratulation *);

PROCEDURE MenueAnzeige;
  BEGIN
                              {  ...  }
  END (* von MenueAnzeige *);

BEGIN (* von Hauptprogramm *)
    MenueAnzeige;
    ReadLn (Anlass);
    IF Anlass = 'Geburtstag'
      THEN GeburtstagsKarte
      ELSE
        IF Anlass = 'Geburtstagsfeier'
          THEN GeburtstagsEinladung
          ELSE
            IF Anlass = 'Klassentreffen'
              THEN KlassentreffenEinladung
              ELSE
                IF Anlass = 'Prüfung bestanden'
                  THEN Pruefungsgratulation
                  ELSE Write (' Unzulässige Eingabe ');
END.
```

Um eine Menü-Auswahl zu programmieren, kann also mit geschachtelten IF-THEN-ELSE-Anweisungen gearbeitet werden.

„Flache" Fallunterscheidung mit ELSE-IF

Hier ist allerdings kein THEN-Teil, sondern nur der jeweilige ELSE-Teil weiter aufgegliedert. Das Programm geht im Grunde nur eine Liste von Möglichkeiten der Reihe nach durch, bis es bei einer zutreffenden angekommen ist. In Worten läßt sich das ganz gut nachvollziehen: „Handelt es sich um den Fall *‚Geburtstag'*? Wenn ja, gib die Karte aus, wenn nein, schaue weiter: Ist es eine *‚Geburtstagsfeier'*? Wenn ja ..." Es liegt also keine tiefe Schachtelung, sondern eine „flache" Ansammlung gleichberechtiger Fälle vor:

```
IF        Anlass = 'Geburtstag'
                        THEN GeburtstagsKarte
ELSE IF Anlass = 'Geburtstagsfeier'
                        THEN GeburtstagsEinladung
ELSE IF Anlass = 'Klassentreffen'
                        THEN KlassentreffenEinladung
ELSE IF Anlass = 'Prüfung bestanden'
                        THEN Pruefungsgratulation
ELSE                         Write (' Unzulässige Eingabe ');
```

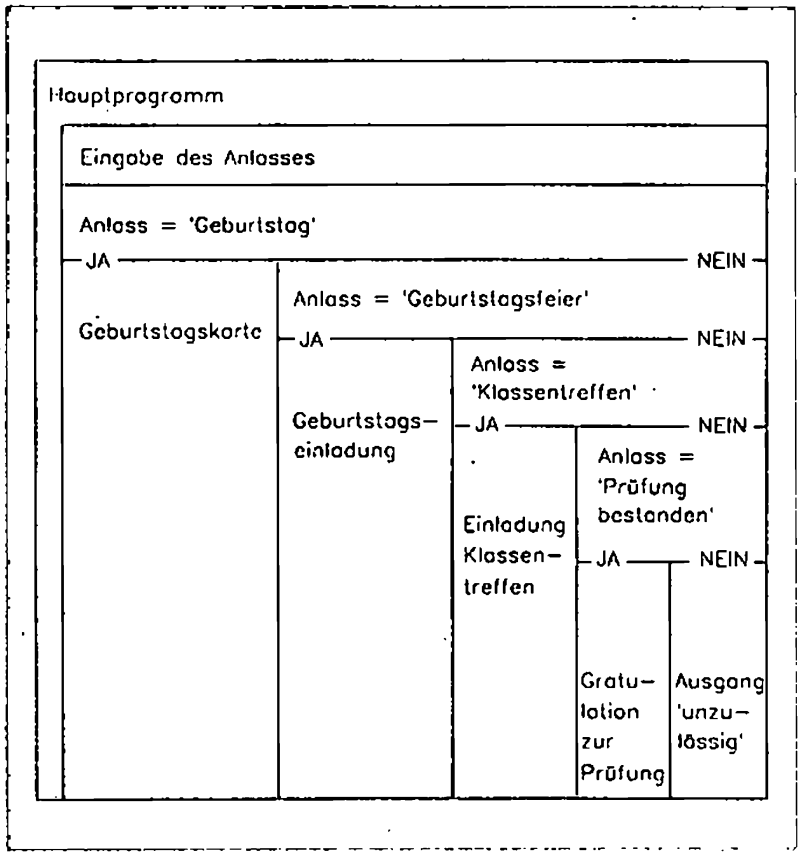

Abbildung 11-1: Fallunterscheidung mit IF-THEN-ELSE

Machen Sie sich auch anhand des Struktogramms diese spezielle Form der mehrfachen Fallunterscheidung klar.

Behandlung von Fehleingaben

Das Programm bedarf aber in mehrfacher Hinsicht noch der Verbesserung. Der erste Punkt betrifft die Korrekturmöglichkeit für unzulässige Eingaben, das heißt für den Fall, daß der eingetippte *Anlass* mit keinem der STRINGs 'Geburtstag', 'Geburtstagsfeier', 'Klassentreffen', 'Prüfung bestanden' übereinstimmt. Bei so langen Wörtern oder Wortkombinationen ist ja die Gefahr des Vertippens relativ hoch. Wenn auf *Anlass* keiner der vorgesehenen Werte steht, erhalten die Benutzer nach der obigen Fassung des Programms nur die Nachricht, daß sie eine „Unzulässige Eingabe" getätigt haben, und das Programm endet. Sie können dann das Programm erneut aufrufen und es nochmal versuchen. Dieses Verfahren ist unbefriedigend, weil ein Programm mindestens in Situationen, in denen Fehler leicht auftreten können, in sich selbst die Gelegenheit zur Wiederholung fehlerhafter Eingaben bieten sollte. Jede solche Korrekturmöglichkeit verlangt die Anwendung einer Schleife, einer Wiederholungsstruktur. Am einfachsten läßt sich dies im Beispiel erreichen, indem die ganze Menü-Auswahl in eine Schleife gesetzt wird.

Die Menü-Schleife

Führen Sie eine boolesche Variable *MenueLaeuft* ein und setzen Sie sie am Anfang des Hauptprogramms auf true. Der bisherige Inhalt des Hauptprogramms dient als Inneres der Wiederholungsstruktur:

```
WHILE MenueLaeuft DO
  BEGIN
                                  { ... }
  END;
```

Damit diese Schleife eine Chance auf ein ordentliches Ende bekommt, fügen Sie dem Menü (am besten als letzten Punkt) die Option „Beenden der Kartenerstellung" hinzu. Entsprechend ergänzen Sie die IF-THEN-ELSE-Schachtelung um den *Anlass* 'Beenden der Kartenerstellung' (in diesem Fall wird die Schleifenkontrollvariable auf false gesetzt). Das Programmstück lautet dann:

```
ELSE
 IF Anlass = 'Prüfung bestanden'
  THEN Pruefungsgratulation
  ELSE
    IF Anlass
       = 'Beenden der Kartenerstellung'
      THEN MenueLaeuft := false
      ELSE Write (' Unzulässige Eingabe ');
```

Programmtest mit Dummy-Prozeduren

Wenn Sie sich nicht die Arbeit machen möchten, die Prozeduren für die Karten im einzelnen auszuführen, so wählen Sie die gleiche Vorgehensweise, die auch bei der Entwicklung größerer Programme angewandt wird: Schreiben Sie sogenannte Dummy-Prozeduren, die nicht vollständig ausgearbeitet sind, sondern im wesentlichen anzeigen, daß sie aufgerufen werden. Eine Dummy-Prozedur *Geburtstagskarte* könnte zum Beispiel so aussehen:

```
PROCEDURE Geburtstagskarte;
  BEGIN
    ClrScr;
    Write (' ******************  Geburtstagsglückwunsch !  ');
    Write (' ******************');
    WriteLn; WriteLn;
    Write ('(Aufruf Prozedur Geburtstagskarte)');
    WriteLn;
  END (* von Geburtstagskarte *);
```

Anregung

Aber vielleicht möchten Sie doch die eine oder andere Prozedur ganz ausführen. Als Anregung mag Abbildung 11-2 auf Seite 169 dienen (für die gepunktete Stelle sollten Sie in der Prozedur eine Eingabemöglichkeit für den Namen vorsehen).

In der Regel sind Menüs ähnlich wie im umgebauten Beispiel in Wiederholungsstrukturen eingebaut. Auf diese Weise wird nicht nur die Korrektur von Fehleingaben möglich, sondern die verschiedenen Optionen können auch mehrfach aufgerufen werden.

Menüs mit Auswahl über Char-Werte

Daß die unterschiedlichen Möglichkeiten, die das Programm bietet, auf dem Bildschirm ausführlich erläutert werden, ist eine gute Sache. Aber für die Auswahl sollte das Tippen möglichst weniger Tasten genügen. Dadurch verringert sich auch die Wahrscheinlichkeit unzulässiger

```
Alles Liebe zu Deinem

  GGG EEEEE BBBB  U   U RRRR  TTTTT  SSSS TTTTT  A     GGG
 G    E     B   B U   U R   R   T   S       T   A A   G
G     E     B   B U   U R   R   T   S       T  A   A G
G  GG EEE   BBBB  U   U RRRR    T    SSS    T  A   A G  GG
G   G E     B   B U   U R   R   T       S   T  AAAAA G
 G  G E     B   B U   U R   R   T       S   T  A   A  G  G
  GGG EEEEE BBBB   UUU  R   R   T   SSSS    T  A   A   GGG

liebe ...............

                                   wünschen Dir
```

Abbildung 11-2: Grafik zur Geburtstagskarte

Optionsaufrufe. Meist sehen daher Menüs so aus, daß den verschiedenen Optionen Nummern (eventuell Funktionstastennummern) zugeordnet sind. Oder ein einzelner, in der Menü-Auflistung hervorgehobener Buchstabe bezeichnet die jeweilige Option. Im Beispiel von oben:

```
D e r   K a r t e n l a d e n

Auswahl:
--------

Glückwunsch zum Geburtstag ..................... 1
Einladung zur Geburtstagsfeier ................. 2
Einladung zum Klassentreffen ................... 3
Gratulation zur bestandenen Prüfung ............ 4
Programm beenden ............................... 0

                                   Ihre Wahl:
```

Abbildung 11-3: Menü zum „Kartenladen", Version 1

Oder:

```
D e r   K a r t e n l a d e n

Auswahl:
--------

Glückwunsch zum Geburtstag ..................... G
Einladung zur Geburtstagsfeier ................. F
Einladung zum Klassentreffen ................... K
Gratulation zur bestandenen Prüfung ............ P
Programm beenden ............................... E

                                   Ihre Wahl:
```

Abbildung 11-4: Menü zum „Kartenladen", Version 2

Menü in Zeilenform

Oft muß mit dem Platz auf dem Bildschirm gegeizt werden. Dann kann eine Menüleiste der zweiten Art auch in ein oder zwei Zeilen statt in Spalten untergebracht werden. Es gilt der erste Buchstabe als Kennzeichen für die jeweilige Option:

```
Geburtstagsglückwunsch, Feier (Einladung),
Klassentreffen (Einladung), Prüfung bestanden, Ende
```

Abbildung 11-5: Menü zum „Kartenladen", Version 3

Bei Kurzeingaben darf die Variable für die Eingabe der Wahl als Integer oder Char genommen werden. Solange keine zweistelligen Zahlen vorkommen, können auch Ziffern zur Kennzeichnung von Menü-Optionen als Char behandelt werden.

Char-Eingabe ohne Bestätigung mit Return

Ist die Menü-Auswahlvariable, im Beispiel *Anlass*, vom Typ Char, genügt dann das Tippen einer einzigen Taste, um die Auswahl im Menü zu vollziehen: Damit eine Char-Variable ohne Bestätigung mit Return eingegeben werden kann, verwenden Sie in Turbo-Pascal bis Version 3.x den KBD-Eingabekanal.

```
Read (kbd, Anlass);
```

Ab Version 4.0 lautet die entsprechende Anweisung (wofür ein USES der Einheit Crt deklariert sein muß):

```
Anlass := ReadKey;
```

Wenn das eingetippte Zeichen auf dem Bildschirm erscheinen soll, muß noch ein *Write (Anlass);* ergänzt werden.

Sobald beim fallweisen Durchgehen einer Liste von Werten mit Integer- oder mit Char-Größen gearbeitet wird, kann die Klarheit der zugrunde liegenden Überlegung und die Übersichtlichkeit des Menüs auch in einer entsprechend übersichtlichen Pascal-Anweisung widergespiegelt werden.

Die CASE-Anweisung

In der CASE-Anweisung (case, „Fall") wird nicht ein einzelner boolescher Ausdruck auf wahr oder falsch (wie bei IF) untersucht. Stattdessen wird eine Liste von möglichen

Werten – die Falliste – auf Übereinstimmung mit dem Wert einer vorgegebenen Variablen beziehungsweise eines vorgegebenen Ausdrucks abgefragt. Im Beispielprogramm kann die IF-THEN-ELSE-Schachtelung gegen folgende CASE-Struktur ausgetauscht werden. Diese hat eine „flache", gut verständliche Form:

```
CASE Anlass OF
  'G': GeburtstagsKarte;
  'F': GeburtstagsEinladung;
  'K': KlassentreffenEinladung;
  'P': Pruefungsgratulation;
  'E': MenueLaeuft := false;
END;
```

Umschrieben heißt dies: Im Fall, daß *Anlass* den Inhalt 'G' hat, führe *Geburtstagskarte* aus, im Fall, daß *Anlass* den Inhalt 'F' hat, rufe *GeburtstagsEinladung* auf, im Fall 'K' *KlassentreffenEinladung* und so fort.

Zusammenfassung mehrerer Fälle

Meist erscheint auf dem Bildschirm der Großbuchstabe als Kennzeichnung einer Menü-Option. Aber der Kleinbuchstabe ist leichter zu tippen, weil dafür die Umschalttaste nicht betätigt werden muß. Am besten wäre es, wenn Drücken des Groß- wie des Kleinbuchstabens denselben Effekt hätten. In der CASE-Anweisung läßt sich dies ganz einfach bewerkstelligen, da auch mehreren (durch Komma getrennten) Zeichen gemeinsam eine Anweisung zugeordnet werden kann. Dies sieht so aus:

```
CASE Anlass OF
  'G', 'g': GeburtstagsKarte;
  'F', 'f': GeburtstagsEinladung;
  'K', 'k': KlassentreffenEinladung;
  'P', 'p': Pruefungsgratulation;
  'E', 'e': MenueLaeuft := false;
 END;
```

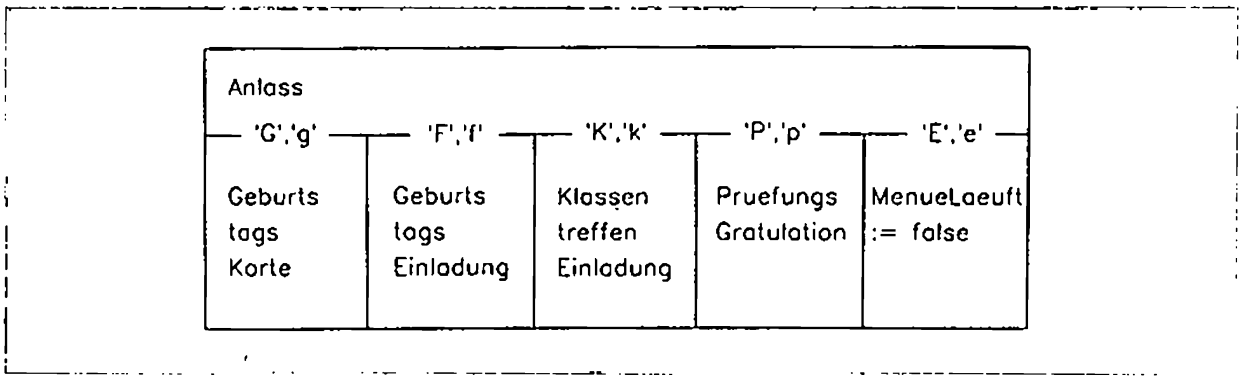

Abbildung 11-6: Struktogramm zur CASE-Anweisung

Struktogramm

Das Struktogramm der CASE-Anweisung beziehungsweise der fallweisen Auswahl bringt gegenüber der gestuften Abfrage mit geschachtelten IF-THEN-ELSE-Anweisungen den gleichen Rang der verschiedenen Fälle deutlich zum Ausdruck.

Syntax der CASE-Anweisung

Die allgemeine Form der CASE-Anweisung lautet:

```
CASE Auswahlausdruck OF
  Werteliste1 : Anweisung1 ;
  Werteliste2 : Anweisung2 ;
  ...
  WertelisteX : AnweisungX ;
END;
```

Das END, das hier steht, schließt die Aufzählung der Fälle in der CASE-Anweisung ab. Es ist ein typisches Beispiel für ein END, dem kein vorausgehendes BEGIN korrespondiert.

Auswahlausdruck mit Operation

Der Auswahlausdruck ist eine Variable, eine Konstante oder ein zusammengesetzter Ausdruck, dessen Wert das Programm noch berechnen muß. Sein Gesamttyp muß ein Aufzählungstyp sein, das heißt in erster Linie Char oder Integer. Nicht zulässig sind Real und STRING. (Wenn zwischen ganzen Wörtern zu unterscheiden ist wie bei der ersten Fassung des Beispielprogramms, kann also die CASE-Anweisung nicht verwendet werden.) Ein Aufzählungstyp ist dadurch gekennzeichnet, daß er eine abgegrenzte Anzahl von Werten umfaßt, die in natürlicher Reihenfolge durchlaufen werden können. Der Computer registriert, welche Fälle behandelt sind und welche nicht. Bei Char geht er die Zeichensatztabelle durch, bei Integer die ganzen Zahlen.

Wie gesagt, kann ein Auswahlausdruck durchaus eine Rechenoperation enthalten. Wenn *Wuerfel1* und *Wuerfel2* Integer-Größen sind, können Sie zum Beispiel folgende Anweisung bilden:

```
CASE Wuerfel1 + Wuerfel2 OF
  2, 4, 6, 8, 10, 12:
  Write ('Gesamtaugenzahl ist gerade');
  3, 5, 7, 9, 11    :
  Write ('Gesamtaugenzahl ist ungerade');
END;
```

Werteliste und Verbundanweisung

Die Werteliste kann einen oder mehrere durch Komma abgetrennte Werte (keine Variablen und keine Rechenausdrücke) enthalten, die alle vom selben Typ wie der Auswahlausdruck sein müssen. Für jede der Anweisungen darf auch eine mit BEGIN-END geklammerte Folge von Anweisungen, eine Verbundanweisung, stehen. Diese sollte dann aber kurz bleiben, da sonst die Übersichtlichkeit der CASE-Anweisung verloren geht.

Haben Sie bemerkt, daß beim Übergang auf die CASE-Anweisung die Meldung bei unzulässigen Eingaben verloren gegangen ist? Das Programm hat, sofern keiner der vorgesehenen Buchstaben getippt wurde, einfach das Menü wiederholt und damit erneut zur Eingabe aufgefordert. Dies muß nicht in allen Pascal-Dialekten so funktionieren. In Standard-Pascal ist eigentlich vorgeschrieben, daß einer der Fälle der CASE-Anweisung zutreffen muß. Wenn der Wert des Auswahlausdrucks keinem Element der Falliste entspricht, soll das Programm mit einer Fehlermeldung abbrechen. Wie mit Werten umgegangen werden kann, die nicht in der Falliste auftauchen, wird jetzt genauer betrachtet.

Die „anderen" Werte bei einer CASE-Anweisung

Bei einer Kette geschachtelter IF-THEN-ELSE-Anweisungen erfaßt der letzte ELSE-Teil alle vorher nicht eigens behandelten Fälle. In vielen Pascal-Dialekten wird, um eine analoge Möglichkeit bei Verwendung einer CASE-Anweisung zu schaffen, so etwas wie ein „Sonst-Fall" als letztes Glied der Falliste zugelassen. In Turbo-Pascal dürfen Sie schreiben:

```
CASE Anlass OF
  'G', 'g': GeburtstagsKarte;
  'F', 'f': GeburtstagsEinladung;
  'K', 'k': KlassentreffenEinladung;
  'P', 'p': Pruefungsgratulation;
  'E', 'e': MenueLaeuft := false
  ELSE  Write (' Unzulässige Eingabe ');
END;
```

Merkzettel

Vorsicht: Setzen Sie keinen Doppelpunkt hinter ELSE!

Ersatz durch eine IF-THEN-ELSE-Konstruktion

Benutzen Sie eine Pascal-Variante, die den „Sonst-Fall" nicht kennt oder möchten Sie so weit möglich ein universelles Pascal schreiben, das nicht auf den Compiler einer bestimmten Firma festgelegt ist, können Sie die CASE-Anweisung in eine einfache Entscheidungsstruktur mit IF-THEN-ELSE einbauen:

```
IF Anlass IN
['G','g','F','f','K','k','P','p','E','e']
  THEN
    CASE Anlass OF
      'G', 'g': GeburtstagsKarte;
      'F', 'f': GeburtstagsEinladung;
      'K', 'k': KlassentreffenEinladung;
      'P', 'p': Pruefungsgratulation;
      'E', 'e': MenueLaeuft := false
    END
  ELSE  Write (' Unzulässige Eingabe ');
```

Einfache Eingabekontrollen und die REPEAT-Schleife

Die Kombination einer IF- mit einer CASE-Anweisung ist nicht sonderlich elegant, weil die zulässigen Fälle praktisch doppelt aufgelistet und beim Programmablauf auch zweimal mit dem Wert von *Anlass* verglichen werden. Ließe sich nicht, anstatt eine Eingabe als unzulässig zu bemängeln und doch mit ihr umzugehen wie mit einem „normalen" Punkt des Menüs, sicherstellen, daß der „Sonst-Fall" nicht eintritt? Mit der folgenden Programmiertechnik verlangen Sie bei unzulässiger Eingabe vom Benutzer direkt eine Korrektur. Die Kontrolle der Eingabe findet in einer Schleifenkonstruktion statt:

```
PROCEDURE MenueEingabe;
CONST
  Fehlermeldung = ' nicht zulässig! ';
VAR
  Zeichen: Char;
BEGIN
  Zeichen := ReadKey;      {In Turbo-Pascal 3.x: Read (kbd,
                           Zeichen);}
  Write (Zeichen);
```

```
  WHILE NOT (Zeichen IN ['G','g','F','f','K','k','P','p','E','e'])
    DO
    BEGIN
      Write (Fehlermeldung);
      Zeichen := ReadKey; {Für 3.x siehe oben}
      Write (Zeichen);
    END;
  Anlass := Zeichen;
END;
```

Es wird ein Zeichen eingelesen und auf den Bildschirm geschrieben. Solange die Eingabe nicht zu den zugelassenen zählt, wird jedesmal eine STRING-Konstante *Fehlermeldung* ausgegeben und Eingabe und Bildschirmecho werden wiederholt. Am Schluß erhält *Anlass* einen gültigen Wert zugewiesen (Sie hätten auch von vorneherein mit der Variable *Anlass* arbeiten können).

Akustische Fehleranzeige

In vielen menügeführten Programmen erscheint keine optische Fehleranzeige. Um störende Zusatztexte auf dem Bildschirm zu vermeiden, läßt man den im Computer eingebauten Lautsprecher mittels spezieller, in den meisten Programmiersprachen vorhandener Funktionen warnende Brumm-, Pfeif- oder Glockentöne erzeugen. Die primitivste Form, einen kurzen Piepser, können Sie schon über den ASCII-Code verwirklichen. Die Ausgabe von Zeichen Nr. 7 erfolgt in der Regel nicht optisch auf dem Bildschirm, sondern akustisch. Probieren Sie folgende Anweisung aus:

```
Write ( chr (7) );
```

REPEAT-UNTIL

Bei der einfachsten, aber durchaus effektiven Form einer Eingabekontrolle wird gar keine Fehlermeldung ausgegeben. Kommentarlos erzwingt das Programm eine erneute Eingabe, bis endlich eine zulässige Taste getippt ist. Dies stellt eine der wenigen Situationen dar, in denen auch eine bisher nicht behandelte Variante der Wiederholungsstruktur, die REPEAT-UNTIL-Schleife, genutzt werden kann.

REPEAT ... UNTIL heißt „wiederhole ... bis" und läßt sich genau so anwenden, wie es die Alltagssprache nahelegt. Zwischen REPEAT und UNTIL können Sie eine oder mehrere Anweisungen schreiben. Die Anweisungsfolge wird wiederholt ausgeführt, bis eine Finalbedingung (Endebedingung) erfüllt ist. Als Finalbedingung fungiert ein boolescher Ausdruck, der hinter UNTIL steht.

Eine kurze Fassung der Eingabeprozedur, die schon das Wesentliche leistet, würde zum Beispiel so aussehen:

```
PROCEDURE MenueEingabe2;
VAR
  Zeichen: Char;
BEGIN
  REPEAT
    Zeichen := ReadKey    {bzw: Read (kbd, Zeichen)}
                {Hier dürfen weitere Anweisungen stehen}
  UNTIL
    Zeichen IN ['G','g','F','f','K','k','P','p','E','e'];
  Write (Zeichen);
  Anlass := Zeichen;
END;
```

Problematik von REPEAT-UNTIL

Die REPEAT-UNTIL-Anweisung läßt an Verständlichkeit nichts zu wünschen übrig, birgt jedoch einen großen Nachteil. Wie ihre Darstellung im Programmablaufplan zeigt, läuft bei ihr der Programmfluß in die zu wiederholende Anweisungsfolge hinein, selbst wenn die Finalbedingung bereits vorher erfüllt ist.

Wenn Sie zum Beispiel einen wiederholten Dateizugriff mit REPEAT-UNTIL programmieren, so würde das Programm einen ersten Leseversuch auch dann starten,

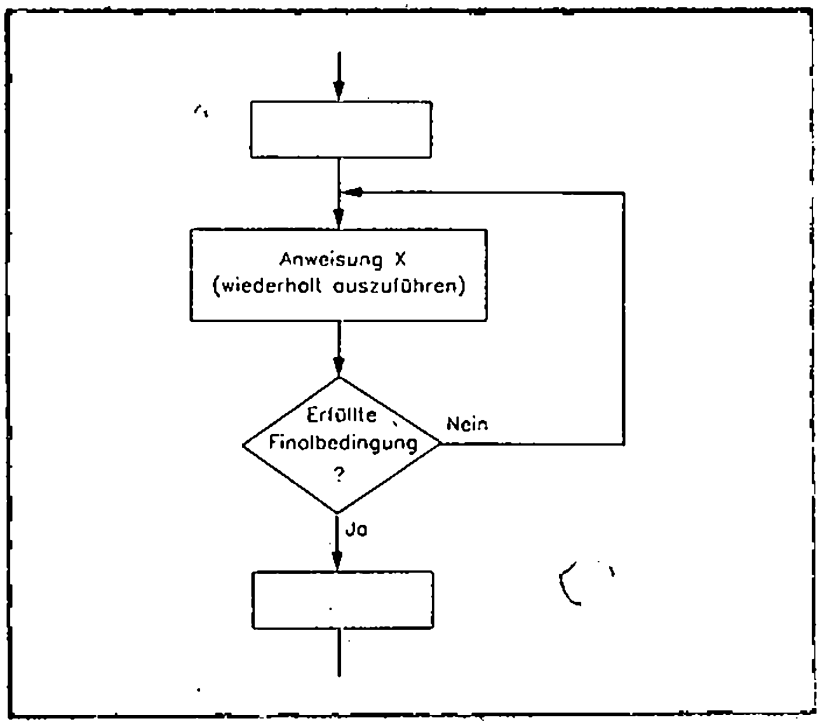

Abbildung 11-7: PAP zur Wiederholungsstruktur mit REPEAT

wenn die eof-Funktion von vorneherein (weil die Datei leer ist) Dateiende signalisiert, und abstürzen. Weil der Fall ungewöhnlich ist, wird er häufig übersehen. Das resultierende Fehlverhalten des Programms trägt einen eigenen Namen: das Null-Fehlersyndrom („Fehler beim 0-ten Durchlauf").

Struktogramm REPEAT-UNTIL

Das Struktogramm der Wiederholungsstruktur mit REPEAT-UNTIL weist deutlich auf den ungeschützten Eingang der Anweisung hin.

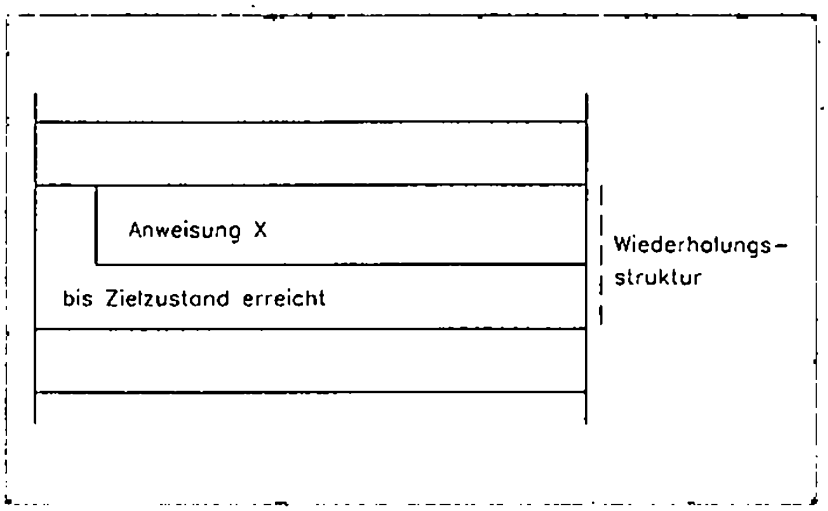

Abbildung 11-8: Struktogramm für Wiederholungsstruktur mit REPEAT

Die gleichwertige WHILE-Konstruktion

REPEAT-UNTIL darf grundsätzlich nur angewandt werden, wenn gewiß ist, daß mindestens ein Schleifendurchlauf stattfinden soll. Im übrigen kann jede REPEAT-UNTIL-Schleife durch eine WHILE-Konstruktion ersetzt werden. Statt:

```
REPEAT Anweisung_X
       UNTIL Finalbedingung_Erreicht;
```

können Sie stets schreiben:

```
Anweisung_X;
WHILE NOT (Finalbedingung_Erreicht) DO
  BEGIN
    Anweisung_X;
  END;
```

Die zweite, syntaktisch gleichwertige Form bedeutet etwas mehr Schreibarbeit, drückt aber unübersehbar aus, daß *Anweisung_X* einmal unabhängig von *Finalbedingung_Erreicht* ausgeführt wird.

Schon bei etwas raffinierten Eingabekontrollen ist die Anwendung einer WHILE-Schleife unerläßlich.

Eine komfortablere Eingabekontrolle

Wenn das Programm in professioneller Manier erlauben soll, falsche Eingaben auf dem Bildschirm zu übertippen, können Sie entweder durchgängig mit GOTOXY arbeiten oder einen kleinen Trick anwenden. Das Zeichen Nr. 8 des ASCII-Codes wird auch Backspace genannt und bewirkt auf dem Bildschirm tatsächlich, daß der Cursor einen Schritt zurück (nach links) wandert. Damit ist die folgende Komfortfassung der Eingabekontrolle programmierbar (die den Trick in drei kleinen Hilfsprozeduren ausnutzt):

```
PROCEDURE MenueEingabe3;
CONST
  Fehlermeldung = ' nicht zulässig! ';
VAR
  Zeichen   : Char;
  i         : Integer;

PROCEDURE CursorEinZeichenNachLinks;
BEGIN
  Write ( chr (8) );
END;

PROCEDURE VorFehlermeldungZurueckgehen;
BEGIN
  FOR i := 1 TO Length (Fehlermeldung) + 1 DO
    CursorEinZeichenNachLinks;
END;

PROCEDURE LoescheFehlermeldung;
BEGIN
  FOR i := 1 TO Length (Fehlermeldung) DO
    Write (' ');
  FOR i := 1 TO Length (Fehlermeldung) DO
    CursorEinZeichenNachLinks;
END;

BEGIN
  Zeichen := ReadKey;      {in Turbo 3.x Read (kbd, Zeichen);}
  Write (Zeichen);
  WHILE NOT (Zeichen IN ['G','g','F','f','K','k','P','p','E','e'])
    DO
```

```
    BEGIN
      Write (Fehlermeldung);
      VorFehlermeldungZurueckgehen;
      Zeichen := ReadKey;      {in Turbo 3.x Read (kbd, Zeichen);}
      Write (Zeichen);
      LoescheFehlermeldung;
    END;
  Anlass := Zeichen;
END;
```

Der Effekt der sichtbaren Überprüfung der Eingabe wird noch deutlicher, wenn als letzte Anweisung in der Schleife ein Verzögerungsbefehl eingefügt wird, etwa: *Delay (200);*.

Die CASE-Anweisung außerhalb von Menüs

Die CASE-Anweisung kann nicht nur bei der Programmierung von Menüs zur übersichtlichen und knappen Darstellung von Mehrfach-Fallunterscheidungen verwendet werden. Auch sonst gibt es nicht selten Situationen, in denen man sich praktisch an einer Liste von Werten orientiert.

Beispiel Druckeranpassung

Angenommen, ein Drucker beherrscht nur den amerikanischen Standard-Druckzeichensatz. Um damit behelfsmäßig einen Text auszugeben, in dem die deutschen Spezialzeichen verwendet wurden, soll ein kleines Anpassungsprogramm entwickelt werden. Seinen Kern kann folgende Schleife bilden, in der mittels einer CASE-Anweisung die Umsetzung der kritischen Zeichen bewerkstelligt wird. (*Datei* muß als Typ *Text* deklariert und vorher eröffnet sein. *Zeichen* hat natürlich den Typ Char.)

```
WHILE NOT eof (Datei) DO
  BEGIN
    Read (Datei, Zeichen);
    CASE Zeichen OF
      'ä'      : Write (Lst, 'ae');
      'ü'      : Write (Lst, 'ue');
      'ö'      : Write (Lst, 'oe');
      'Ä'      : Write (Lst, 'Ae');
      'Ö'      : Write (Lst, 'Oe');
      'Ü'      : Write (Lst, 'Ue');
```

```
      'ß'        : Write (Lst, 'ss')
      ELSE     Write  (Lst, Zeichen)
    END;
  END;
```

Für den ernsthaften Einsatz dieser Anweisungen empfiehlt sich übrigens, auch keine anderen Zeichen aus der zweiten Hälfte der ASCII-Tabelle durchzulassen (die Zeichen würden wahrscheinlich Chaos auf dem Drucker anrichten). Dafür müssen Sie lediglich das Write im ELSE-Teil der CASE-Anweisung noch einer entsprechenden Bedingung unterwerfen. Mit der *ord*-Funktion können Sie die ASCII-Nummer des gelesenen Zeichens feststellen:

```
IF ord (Zeichen) < 127
  THEN Write (Lst, Zeichen)
```

Werteabschnitte in der Falliste

Wenn in der CASE-Anweisung ein ganzer Abschnitt von Werten, etwa alle „normalen" Kleinbuchstaben, als einziger Fall behandelt werden soll, müssen Sie die Buchstaben nicht alle einzeln hinschreiben. Sie bilden den Fall „von 'a' bis 'z'", indem Sie zwei Punkte zwischen Anfangs- und Schlußwert des Abschnitts setzen. Unter Abschnitt ist dabei ein ununterbrochenes Teilstück aus der Werteliste des zugrundegelegten Aufzählungstyps, zum Beispiel der ASCII-Zeichensatztabelle, zu verstehen:

```
CASE Zeichen OF
  'ä'       : Write ('ae');
  'ü'       : Write ('ue');
  'ö'       : Write ('oe');
  'ß'       : Write ('ss');
  'a'..'z'  : Write (Zeichen);
                              { ... }
END;
```

Beispiel: CASE-Anweisung mit Integer-Auswahl-Ausdruck

Für ein weiteres kleines Beispiel einer CASE-Anwendung, in der diesmal mit Integer-Werten gearbeitet wird und ein Rechenausdruck als Auswahlkriterium dient, steht das folgende Programm. Es verlangt die Eingabe des Wochentags, auf den der Monatserste fällt, und kann dann zu einem beliebigen Tag des betreffenden Monats den Wochentag in Klartext ausgeben.

```
PROGRAM KleinerKalender;

USES crt, printer;
```

```
VAR
  ErsterTag     : Integer;
  FraglicherTag : Integer;

BEGIN
  ClrScr;
  WriteLn ('Geben Sie den Wochentag des Monatsersten ein ');
  Write
  ('(Mo = 1, Di = 2, Mi = 3, Do = 4, Fr = 5, Sa = 6, So = 7) ');
  ReadLn (ErsterTag);
  WriteLn;
  WriteLn ('Geben Sie als ganze Zahl ein, von welchem Tag Sie ');
  Write   ('den Wochentag wissen wollen:   ');
  ReadLn  (FraglicherTag);
  WriteLn;
  CASE (ErsterTag + FraglicherTag - 1) MOD 7 OF
      1 : WriteLn ('Der ', FraglicherTag, '. ist ein Montag.');
      2 : WriteLn ('Der ', FraglicherTag, '. ist ein Dienstag');
      3 : WriteLn ('Der ', FraglicherTag, '..ist ein Mittwoch');
      4 : WriteLn ('Der ', FraglicherTag, '. ist ein Donnerstag');
      5 : WriteLn ('Der ', FraglicherTag, '. ist ein Freitag');
      6 : WriteLn ('Der ', FraglicherTag, '. ist ein Samstag');
      0 : WriteLn ('Der ', FraglicherTag, '. ist ein Sonntag.');
    END;
END.
```

Auswertung von Scancodes

Eine weitere Anwendung der CASE-Anweisung bildet die Auswertung von Funktions- und Spezialtasten. Wie in Kapitel 7 erwähnt, liefert ab Turbo-Pascal 4.0 sowie in Quick-Pascal die ReadKey-Funktion bei Betätigung einer dieser Tasten zunächst die ASCII-Nummer 0. Mit einer weiteren Abfrage des Tastaturkanals erhalten Sie den sogenannten Scancode als ASCII-Zeichen. Zum Beispiel können Sie auf folgende Art feststellen, welche Cursortaste betätigt wurde (*Eingabe* muß als Char-Variable deklariert sein):

```
Eingabe : = ReadKey;
IF Eingabe = #0
   THEN
     BEGIN
       Eingabe : = ReadKey;
       CASE Eingabe OF
         #75: {Cursor nach links wurde gedrückt};
         #77: {Cursor nach rechts gedrückt};
         #72: {Cursor nach oben};
```

```
        #80: {Cursor nach unten}
      END
    END;
```

Den Scancode einer Taste oder Tastenkombination sagt Ihnen das folgende winzige Programm (zum Verständnis beachten Sie, daß mit jedem Aufruf von ReadKey ein weiteres Zeichen aus dem Tastaturkanal geholt wird):

```
PROGRAM ScanCode;
   USES CRT;
BEGIN
     IF Ord (ReadKey) = 0 THEN
        WriteLn ('Taste  ', Ord (ReadKey));
END.
```

Das nächste Kapitel zeigt, wie Fallisten samt den erforderlichen Werten konstruiert werden können.

Aufgaben

1. Versuchen Sie, das Syntaxdiagramm der CASE-Anweisung zu zeichnen!

2. Schreiben Sie das Programm zu Ende, das bei einem auf Diskette vorliegenden Text Umlaute und 'ß' in die entsprechenden Buchstabenkombinationen auflöst.

3. Ergänzen Sie das Programm *KleinerKalender* so, daß nach Eingabe des Monats abgeprüft wird, ob es den angegebenen Tag im betreffenden Monat überhaupt gibt (bekanntlich hat der Januar 31 Tage, Februar – außer im Schaltjahr – 28, März 31 und so weiter).

4. Übersetzen Sie die folgende Anweisung formal in eine gleichwertige Kombination einer REPEAT-Anweisung mit einer Entscheidungsstruktur:

```
WHILE NOT Finalbedingung_Erreicht DO
  Anweisung_X;
```

12 Aufzählungs-, Teilbereichs- und SET-Typen

Werte, Variable, Typen – Zur allgemeinen Syntax der Typdeklaration – Selbstdefinierte Aufzählungstypen – Teilbereichstypen – SET-Typen – Aufgaben

Ein Programm „modelliert" einen Ausschnitt aus der Wirklichkeit. Es soll einen Realitätsbereich in einem formalen System widerspiegeln. Die (relative) Stimmigkeit der Modellkonstruktion hängt grundlegend von der Definition passender Variablen und des richtigen Zuschnitts ihrer Typen ab: bei Personendatenverarbeitung wie bei Spielprogrammen, für Grafiksysteme wie für Textverarbeitung. Pascal zeichnet sich durch seine reichhaltigen Möglichkeiten zu eigenen Definitionen von Datentypen aus. Die Kreation sinnvoller Datentypen entlastet dann auch die Detailprogrammierung von nicht dorthin gehörenden Problemen. Es erweist sich immer wieder, daß Strukturfragen in der Programmierung auf die Datenmodellbildung, die richtige Typenkonstruktion zurückzuführen sind. Nicht zuletzt wird mit einem durchdachten Typenkonzept auch dem Prinzip Rechnung getragen, daß Bedeutungen sich in Text und Aufbau des Programms selbst niederschlagen sollen.

Werte, Variable und Typen

Fertig vorliegende Typen

Die elementaren (einfachen) Standardtypen Char, Integer, Real und Boolean wurden bereits beschrieben. Eine Variable vom Typ Char hat als mögliche Werte die Zeichen des Zeichensatzes. Eine Integer-Variable kann mit ganzen Zahlen belegt werden, eine Real-Variable mit Dezimalzahlen. Boolesche Größen tragen die booleschen Werte true oder false. Außerdem haben Sie den vordefinierten Spezialtyp STRING und den Standardtyp *Text* verwendet. Je nach Pascal-Dialekt gibt es noch eine unterschiedliche Zahl anderer solcher gebrauchsfertiger Typen, die ohne weiteren Aufwand zur Verfügung stehen.

Es kann hier zwischen *Standardtypen*, das heißt Typen, die vom jeweiligen Pascal-System entsprechend den Vorschriften von Standard-Pascal bereitgestellt werden, und *allgemeinen vordefinierten Typen* unterschieden werden.

Drei Möglichkeiten, Typen selbst zu kreieren

Um der Universalität möglicher Anwendungen gerecht zu werden, können und sollen Sie sich in Pascal zusätzlich eigene Variablentypen schaffen. In einer Reihe von Situationen werden Sie sogar zu eigenen Typdefinitionen gezwungen. Grundsätzlich gibt es drei Möglichkeiten, neue Typen zu kreieren:

1. Sie können sich eigene neue Reihen von elementaren Werten ausdenken, analog der Zeichensatztabelle beim Typ Char beziehungsweise der Alternative false, true, die den Typ Boolean ausmacht. Auf diese Weise erhalten Sie einen selbstdefinierten Aufzählungstyp.

2. Aus der geordneten Wertefolge eines Aufzählungstyps können Ausschnitte als Teilbereichs- oder Ausschnittstypen vereinbart werden.

3. Das bei weitem wichtigste Mittel der Typenschöpfung gilt „großen" Objekten, die nicht mit elementaren Typen beschrieben werden können. In Pascal können Sie auf diverse Arten einen neuen Typ aus einfacheren Typen kombinieren, um zum Beispiel den Zusammenhang der auf eine Person oder ein Kartenspiel bezogenen Daten auszudrücken. Ein solcher Datentyp heißt dann „zusammengesetzt", „komplex" oder „strukturiert".

Am Schluß dieses Kapitels finden Sie die SET-Typen erklärt, das sind die noch am wenigsten kombinationsfreudigen unter den zusammengesetzten Datentypen. In der Hauptsache werden die strukturierten Typen in Kapitel 14 behandelt. Das vorliegende Kapitel ist vor allem den ersten beiden Arten der Typenkreation gewidmet, an deren Beispiel Sie ein erstes Verständnis des für gutes Programmieren grundlegenden Typkonzepts in Pascal entwickeln sollen.

Typ, Variable, Werte

Der Zusammenhang von Typ, Variablen und Werten läßt sich in einem Satz formulieren: Der Typ einer Variablen bestimmt die Werte, welche die Variable annehmen kann.

Mit einem Begriff aus der Mathematik ausgedrückt, entspricht der Typ dem Definitionsbereich der Variablen. Das bedeutet, er umfaßt alle Werte, die auf die Variable mittels Wertzuweisung gesetzt oder in einer Eingabe verwendet werden dürfen. Das folgende Diagramm skiz-

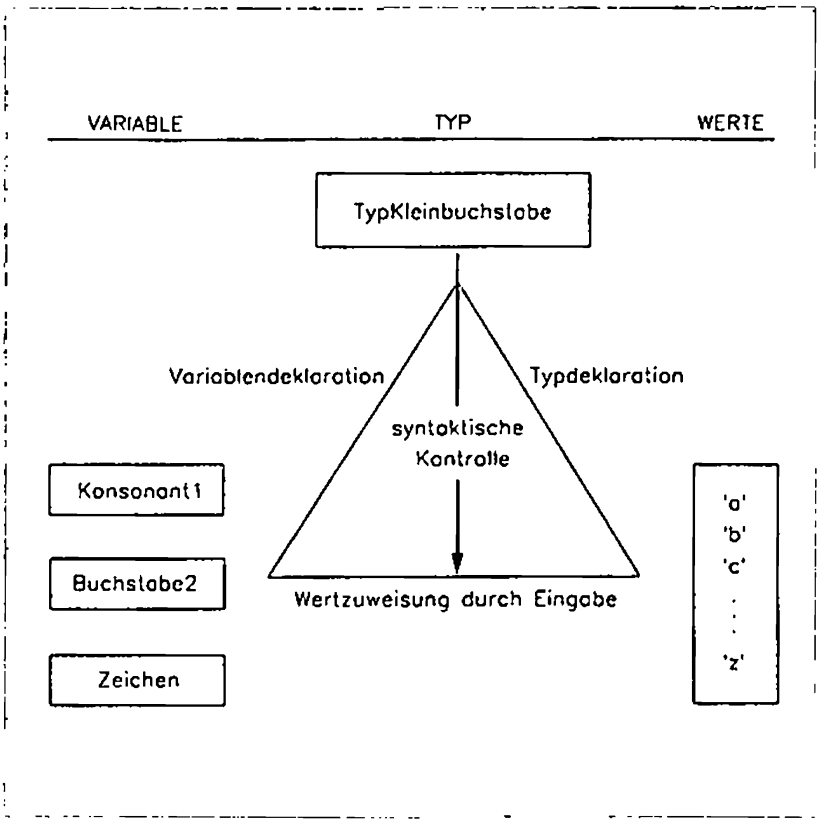

Abbildung 12-1: Variable – Typen – Werte

ziert das Dreiecksverhältnis von Variablen, Werten und ihrem Typ:

In der Variablendeklaration erklären Sie, welchem Typ eine (variable) Größe zuzurechnen ist. Mittels Wertzuweisung oder Eingabe wird einer Variablen ein Wert, der laut ihrem Typ zulässig ist, zugewiesen.

Welche Werte es sind, die zum jeweiligen Typ gehören, ist bei vordefinierten Typen dem Handbuch der Pascal-Implementation zu entnehmen. Bei einem selbstdefinierten Typ geben Sie im Programmtext das Spektrum der Werte selbst an, für die der Typ steht. Dies geschieht in einer sogenannten Typdeklaration (oder Typdefinition). Zum Beispiel kann der *TypKleinbuchstabe* definiert werden durch (genauere Erklärung folgt unten):

```
TYPE
  TypKleinbuchstabe = 'a' .. 'z';
```

Formale Prüfkriterien durch Typen

Entscheidend ist, daß der Typ eine formale, syntaktische Bestimmung der zulässigen Werte einer oder mehrerer Variablen gestattet. Das Pascal-System kann dadurch automatisch kontrollieren, ob ein bestimmter Wert zu einer Variable paßt.

Bei Wertzuweisungen, Vergleichsausdrücken und in anderen Zusammenhängen erkennt eventuell schon der Compiler eine Inkompatibilität. Von vorneherein wird verhindert, daß einer Variablen ein nicht mit ihr verträglicher Ausdruck zugewiesen wird. Der Compiler bemängelt zum Beispiel (in Turbo-Pascal mit der Fehlermeldung *Type mismatch*), wenn Sie auf eine STRING-Variable einen booleschen Wert oder auf eine Integer-Größe einen Real-Wert setzen. Durch die Weigerung, ein lauffähiges Programm abzuliefern, schützt der Compiler Sie vor möglicherweise verheerenden Folgen. Zum Beispiel würden bei der Zuweisung eines Real-Wertes an eine Integer-Variablen die überzähligen Stellen der Real-Zahl verloren gehen. Übrigens gibt es auch Compiler, vor allem bei Sprachen mit weniger strikter Syntax als Pascal, die bei leichten Fehlern zwar warnen, aber trotzdem zu Ende kompilieren.

Eine festgestellte Typ-Inkompatibilität weist immer auf einen logischen Fehler im Programmcode hin. Wer versucht, den sprichwörtlichen Vergleich von Äpfeln und Birnen durchzuführen, wird vom Compiler daran gehindert. Dazu ist die Software imstande, weil sie die formale Information ausnutzt, daß die eine Variable zum Beispiel mit *TypApfel*, die andere mit *TypBirne* deklariert ist. Letztlich werden Sie auf den Widerspruch zu Ihrer eigenen vorherigen Erklärung (Deklaration) aufmerksam gemacht, laut der es sich um unterschiedliche Dinge handelt. Die Anwendung von Typen zeigt sich damit als ein Weg, um eine semantische Eigenschaft des Programms – einen Aspekt der Programmbedeutung – mit syntaktischen, das heißt formallogischen Mitteln widerzuspiegeln. Sie liegt in der Linie dessen, was Programmentwicklung bedeutet: Realität in formalen Systemen abbilden.

Die Typdeklarationen im Gesamtaufbau eines Pascal-Programms

Die Definitionen (Deklarationen) eigener Typen stehen im Deklarationsteil eines Programms oder einer Prozedur laut den Vorschrift von Standard-Pascal *nach* den Konstanten und *vor* den Variablen (die wiederum vor den untergeordneten Prozeduren kommen). Diese Reihenfolge ist logisch, weil die Typen für die Deklaration von Variablen zur Verfügung stehen sollen, andererseits in ihre Definition Konstanten eingehen können. In Turbo- und Quick-Pascal dürfen die verschiedenen Abschnitte des

Deklarationsteils auch mehrfach und in gemischter Folge vorkommen. Natürlich darf trotzdem ein Pascal-Name nicht zweimal deklariert sein. Und jeder Typ muß existieren, bevor er zur Vereinbarung von Variablen dienen kann. Schematisch gliedert sich insgesamt ein Programm (analog eine Prozedur) in Standard-Pascal entsprechend dem folgenden Schema:

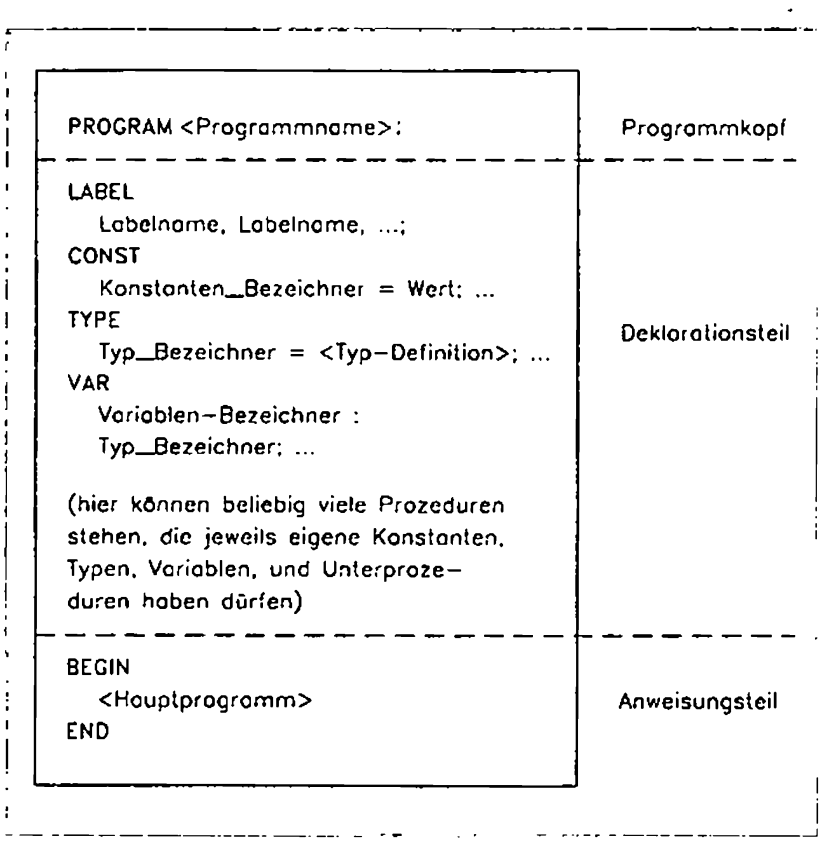

Abbildung 12-2: Schema eines Pascal-Programms

Nach Standardsyntax gibt es keine weiteren Abschnitte eines Programms beziehungsweise einer Prozedur. Die bislang nicht erwähnten LABEL-Deklarationen sind Vereinbarungen für Sprung-Marken. Aus den im Abschnitt über GOTO (Kapitel 9) dargelegten Gründen sollten sie sehr zurückhaltend gebraucht werden und werden in diesem Buch nicht behandelt. In Turbo-Pascal ab Version 4.0 und in Quick-Pascal steht, wie Sie gelesen haben, meist hinter dem Programmkopf eine USES-Deklaration für eine oder mehrere Units, vorcompilierte Programmeinheiten (oder Moduln).

Zur allgemeinen Syntax der Typdeklaration

Unter dem Titel TYPE können beliebig viele Typdeklarationen (Typdefinitionen) notiert werden. Dabei hat eine Typdeklaration die allgemeine Gestalt:

```
Typname = Typbeschreibung;
```

Ebenso wie in Konstantendefinitionen ist hier das einfache Gleichheitszeichen verlangt (nicht := und auch nicht :). Typname darf irgendein nach den üblichen Regeln gebildeter Pascal-Name sein. Im einfachsten Fall besteht die Typbeschreibung aus dem Namen eines bereits definierten Typs:

```
ApfelgewichtsTyp = GewichtsTyp;
```

In dieser Zeile ist *ApfelgewichtsTyp* der neu vereinbarte Name. Was rechts steht, der *GewichtsTyp*, muß vorher bekannt sein.

Es empfiehlt sich, grundsätzlich jeden Typnamen mit „Typ" beginnen oder enden zu lassen, obgleich ein beliebiger Pascal-Name erlaubt ist. Sonst verliert sich ganz schnell der Überblick, was als Typname und was als Name einer Variablen gedacht war. Im Beispiel muß also *GewichtsTyp* vorher deklariert werden (etwa durch: *GewichtsTyp = Real;*). Damit wird *GewichtsTyp*, indirekt auch *ApfelgewichtsTyp*, auf den vordefinierten Typ Real zurückgeführt. Es erscheint nur auf den ersten Blick wie eine lästige Definitionsübung, für syntaktisch denselben Typ (hier Real) weitere Bezeichnungen einzuführen. Will man später irgendwann die Variablengruppen unabhängig voneinander verändern, ist es durchaus sinnvoll, wenn die Typen von vorneherein getrennt gehalten werden. Selbst wo es syntaktisch zunächst unerheblich ist, kann die Zuordnung der Variablen zu unterschiedlichen Typen bei der Programmentwicklung helfen. Aufgrund obiger Typdefinitionen können im VAR-Abschnitt zum Beispiel folgende Größen vereinbart werden:

```
PausenapfelGestern, PausenapfelHeute,
EinApfel, Durchschnittsgewicht          :  ApfelgewichtsTyp;
MappenGewichtGestern, MappenGewichtHeute:  GewichtsTyp;
```

Nicht ganz so einfach sind die Typbeschreibungen, wenn tatsächlich neue Typen geschaffen werden. Zu-

nächst sollen die selbstdefinierten Aufzählungstypen betrachtet werden.

Selbstdefinierte Aufzählungstypen

Möchten Sie einen eigenen Aufzählungstyp kreieren, müssen Sie eine charakteristische und geordnete Liste von Werten selbst angeben. Die Typbeschreibung besteht aus einer Aufzählung von Pascal-Namen, die in Klammern gesetzt werden. In einem Programm zur Verwaltung einer Apfelzucht könnten zum Beispiel Variable und damit ein Typ für Apfelsorten von Nutzen sein. Die Definition des Typs sähe unter der Überschrift TYPE vielleicht so aus:

```
TypApfelsorte = (Jonathan, Klarapfel, Boskop, CoxOrange,
                 SommerCox, GoldenDelicious, GrannySmith,
                 Ontario, Renette);
```

Entsprechende Variablen könnten unter der Überschrift VAR deklariert werden, etwa:

```
Lieblingssorte, Apfelsorte, VerkaufSorte: TypApfelsorte;
```

Wertzuweisungen würden so aussehen:

```
Lieblingssorte := Jonathan;
VerkaufSorte   := CoxOrange;
```

Für anschauliche kleine Beispiele eignet sich auch gut der *TypWochentag*:

Beispiel TypWochentag

```
TYPE
TypWochentag = (Mon, Die, Mit, Don, Fre, Sam, Son);
```

Vorsicht, Donnerstag kann nicht als „Do" abgekürzt werden, weil DO als Bestandteil der WHILE- und der FOR-Anweisung ein pascaleigenes Schlüsselwort darstellt! Computerintern wird übrigens die Liste jedes Aufzählungstyps mit 0 beginnend durchnumeriert, das heißt, es besteht die Zuordnung:

Mon	Die	Mit	Don	Fre	Sam	Son
0	1	2	3	4	5	6

Wenn der Typ vereinbart ist, können wieder Variablen dieses Typs deklariert werden, also zum Beispiel:

```
VAR
tag, heute, morgen, gestern :  TypWochentag;
```

Anwendung von selbstdefinierten Aufzählungstypen

Kann *TypWochentag* vielleicht vorteilhaft in dem Programm *KleinerKalender* vom Schluß des letzten Kapitels verwendet werden? Leider in der Praxis nicht, denn alle Anwendungen von selbstdefinierten Aufzählungstypen leiden unter einem Handicap: Eine unmittelbare Ein- oder Ausgabe der Werte von Variablen eines selbstdefinierten Aufzählungstyps mit Write oder Read ist nicht möglich. Dies ist unangenehm, aber nicht unbegründet, da Ein- und Ausgabe im Grunde immer Textdateien ansprechen (Tastatur und Bildschirm sind Spezialfälle). Eigentlich können nur Zeichen empfangen und abgegeben werden. Schon bei Zahlen muß das Pascal-System dafür sorgen, daß bei Ein- und Ausgabe die Daten zwischen der Char- beziehungsweise STRING-Darstellung und dem internen Zahlenformat umgewandelt werden. Bei der Eingabe müssen die Zahlen syntaktisch geprüft, bei der Ausgabe formatiert werden. Der Aufwand, entsprechende Mechanismen für alle denkbaren selbstdefinierten Typen im Pascal-System mitzuliefern, wurde vermieden.

Darstellung einer logischen Struktur

Praktisch dienen die eigenen Aufzählungstypen in erster Linie dafür, in größeren Programmen eine innere Struktur übersichtlich darzustellen. Wenn zum Beispiel ein Programm je nach Wochentag Unterschiedliches tun soll, kann die wie oben vereinbarte Variable *heute* samt den Werten ihres Typs in einer CASE-Anweisung verwendet werden (mit entsprechenden Prozeduren *Werktagsfahrplan*, *Samstagsfahrplan*, *Sonntagsfahrplan*):

```
CASE heute OF
  Mon..Fre : Werktagsfahrplan;
  Sa       : Samstagsfahrplan;
  So       : Sonntagsfahrplan;
END;
```

Durchlauf mit der FOR-Anweisung

Wichtig ist auch, daß die FOR-Anweisung erlaubt, die Werte eines Aufzählungsstyps oder eines Teilbereichs der Reihe nach abzuprüfen:

```
FOR heute  := Mon TO Fre DO
  DruckeTerminzettel;
```

Nehmen Sie an, das Beispielprogramm muß für verschiedene Zwecke die ganze Liste der Apfelsorten durchgehen. Der oben erklärte *TypApfelsorte* entfaltet seine Vor-

züge besonders, wenn er in einer FOR-Anweisung verwendet wird. Das Programm könnte (mit obigen Deklarationen, dazu einer Integer-Variable *Anzahl* und den Real-Variablen *MittlererPreis* und *Einzelpreis*) den Durchschnittspreis je Kilogramm Äpfel mit folgenden Anweisungen ermitteln (dabei sei angenommen, daß der Preis für die einzelnen Sorten passend eingetippt wird):

```
MittlererPreis := 0;
Anzahl         := 0;
FOR Apfelsorte := Jonathan TO Renette DO
  BEGIN
    Write (' nächster Wert: .');
    ReadLn (Einzelpreis);
    MittlererPreis := MittlererPreis + Einzelpreis;
    Anzahl         := Anzahl + 1;
  END;
MittlererPreis := MittlererPreis / Anzahl;
WriteLn;
WriteLn (' Der durchschnittliche Preis ist',
           MittlererPreis : 7 : 2);
```

Diese Anweisungen können unverändert weiter benutzt werden, wenn zusätzliche Apfelsorten in die Mitte der den Typ beschreibenden Werteliste hinzukommen, weil nur Anfangs- und Endwert namentlich angeführt sind. Für diesen Vorteil kann es sich in einem größeren Programm, das die Liste der Apfelsorten häufig abarbeitet, durchaus lohnen, einen höheren Aufwand für die Programmierung der Datenein- und ausgabe zu treiben.

Eine Ausgabeprozedur für einen selbstdefinierten Aufzählungstyp

Wenn in dem Beispiel die Namen der Sorten angezeigt werden sollen, bleibt nicht viel anderes übrig, als eine gesonderte Prozedur *AusgabeApfelsorte* zu schreiben. Dort müssen in einer CASE-Anweisung alle Werte des Typs noch einmal aufgelistet und individuell in Ausgaben umgesetzt werden:

```
PROCEDURE AusgabeApfelsorte;
BEGIN
  CASE Apfelsorte OF
    Jonathan        : Write ('Jonathan');
    GoldenDelicious: Write ('Golden Delicious');
                          { .... }
  END;
END;
```

Im nächsten Kapitel wird beschrieben, wie eine solche Prozedur so eingerichtet werden kann, daß sie für sämtliche Variablen des *TypApfelsorte* anwendbar ist.

Ord und die Funktionen bei Aufzählungstypen

Die Funktion *ord* läßt sich bei selbstdefinierten Aufzählungstypen genauso verwenden wie bei Char-Werten. Die Ordnungsnummer eines Wertes ergibt sich aus dem Platz des Wertes in der den Typ definierenden Liste. Bei *TypWochentag* oben hat:

```
ord (Mon)        {den Wert 0,}
ord (Die)        {den Wert 1 und so weiter}
```

Vergleichsoperationen

Die Anordnung der Werte kann bei allen Aufzählungstypen wie bei Zahlen mit >, >=, =, <=, < abgefragt werden. Zum Beispiel gilt für die Werte von *TypWochentag*: *Mon* < *Die*, *Die* < *Mit* und so weiter.

Vorgänger und Nachfolger

Bei allen Aufzählungstypen gibt es auch die Möglichkeit, den Nachfolger, successor, und den Vorgänger, predecessor, eines bestimmten Wertes (entsprechend der durch die Typdefinition festgelegten Reihenfolge) zu ermitteln: *succ (Die)* zum Beispiel hat den Wert *Mit*, *pred (Mit)* gibt wieder *Die*.

Für eine Integer-Größe *Zahl* kann *succ (Zahl)* und *pred (Zahl)* ebenfalls gebildet werden. *succ (Zahl)* bedeutet dasselbe wie *Zahl + 1*, *pred (Zahl)* ist *Zahl – 1*. Unter Umständen werden die Nachfolger- und Vorgängerfunktion auf dem Computer sogar schneller ausgewertet als die Addition und Subtraktion. Abschließend noch ein Beispiel von *TypWochentag*:

```
IF heute < Son
  THEN morgen := succ (heute)
  ELSE morgen := Mon;
```

Teilbereichstypen

Der Grundgedanke im Zusammenspiel von Typen, Variablen und Werten lautet, wie oben geschildert:

Der Typ gibt syntaktisch die möglichen, semantisch die sinnvollen Werte einer Variablen an. Pascal erlaubt, dieses Prinzip noch etwas strikter anzuwenden, indem es anbietet, zu Aufzählungstypen Teilbereichs- oder Aus-

schnittstypen zu bilden. Wenn zum Beispiel eine Variable des Wochentagtyps nur für Werktage verwendet werden soll, kann ein entsprechender Teilbereich vereinbart werden. Das sieht dann so aus:

```
TYPE
TypWerktag   =  Mon .. Sam;
VAR
Werktag      : TypWerktag;
```

Auch für Integer und Char können entsprechende Ausschnittstypen und zugehörige Variablen definiert werden:

```
TYPE
TypKleinbuchstabe  = 'a'..'z';
TypJahrNachChristi = 1 .. 2000;
TypJahrVorChristi  = -30000 .. -1;
VAR
Zeichen  :  TypKleinbuchstabe;
JahrNChr :  TypJahrNachChristi;
JahrVChr :  TypJahrVorChristi;
```

In Turbo-Pascal gibt es sogar einen vordefinierten Teilbereichstyp von Integer, den Typ *Byte* für ganze Zahlen zwischen 0 und 255. Um diesen Typ selbst zu deklarieren, müßte man schreiben: **Byte**

```
TYPE Byte = 0 .. 255;
```

Der *vordefinierte* Typ Byte entspricht tatsächlich einem Byte im Speicher. Ein *selbstdefinierter* Typ übernimmt aber in Turbo-Pascal die Speicherform seines Grundtyps. Eine Variable des eigenen Typs Byte würde also den gleichen Platz wie eine Integer-Variable, das heißt zwei Byte, belegen.

Es ist oft nützlich, folgenden Teilbereichstyp des Typs Integer zu deklarieren: **Cardinal**

```
TYPE  Cardinal = 0..MaxInt;
```

Kardinalzahlen heißen die natürlichen Zahlen, die eine Anzahl feststellen (0, 1, 2, 3...). Demgemäß umfaßt *Cardinal* die ganzen Zahlen zwischen 0 und der maximalen Integer-Zahl. Dieser Typ kann sinnvoll bei Zählvariablen verwendet werden. Ab Version 4.0 gibt es in Turbo-Pascal zum Zählen den vordefinierten Typ *Word*. Im Gegensatz zu einem Teilbereichstyp von Integer kann er zwei Bytes im positiven Bereich voll ausnutzen und damit Zählzahlen von 0 bis 65535 zur Verfügung stellen. **Word**

Beschränkte Verwendbarkeit von Teilbereichstypen

Ein Teilbereichstyp wird, wie an den Beispielen zu sehen, folgendermaßen definiert: Die Typbeschreibung beginnt mit dem ersten Wert des gewünschten Ausschnitts vom Grundtyp, gefolgt von zwei Pünktchen und dem letzten zugehörigen Wert. Aus dieser einzig möglichen Definitionsweise ergibt sich die wesentliche Einschränkung im Gebrauch von Teilbereichstypen: Nur vollständige Abschnitte des Grundtyps in seiner ursprünglichen Ordnung können zu einem eigenen Typ gemacht werden. Es kann aber durchaus vorkommen, daß Variable benötigt werden, deren Werte in unzusammenhängenden Abschnitten des Grundtyps liegen. Zum Beispiel wurde oben *TypKleinbuchstabe* definiert. In der aufgeführten Form eignet er sich nur für Programme, die auf amerikanische Verhältnisse zugeschnitten sind. Sollen deutsche Texte bearbeitet werden, müßten noch 'ä', 'ö', 'ü', 'ß' hinzugenommen werden. Aber das geht mit einem Teilbereichstyp (unter Zugrundelegung des PC-Zeichensatzes) nicht: das Herausgreifen zum Beispiel aller Kleinbuchstaben aus dem Alphabet überstrapaziert das Pascal-Typkonzept.

Für die Feinunterscheidung von Werten innerhalb eines Grundtyps sollte das Programm mit eigenen Mitteln sorgen. Es muß auch sonst auf unterschiedliche Weisen, zum Beispiel mit Eingabenkontrollen, verhindern, daß unzulässige Werte auf eine Variable kommen. Das oben formulierte Prinzip, daß die Typen eine syntaktische Prüfung der Zuweisungen an Variable erlauben, ist eher als Notbremse bei groben Fehlern zu verstehen. Zur regulären Fehlerverhütung taugt die Typkontrolle des Pascal-Systems vor allem in jenen Fällen wenig, in denen sie erst während des Programmlaufs greift und dabei einen – mindestens bei größeren Programmen nach Möglichkeit zu vermeidenden – Absturz herbeiführt. Die wichtigsten Anwendungen von Teilbereichstypen sind Typen wie Byte oder Cardinal, die auch technisch unterschiedlich behandelt werden können. Ansonsten liegt ihr Hauptzweck weniger in der Verfeinerung der Syntaxkontrolle als darin, daß sie in die Definition bestimmter zusammengesetzter Datentypen eingehen, wie Sie unten sehen werden.

Explizite und implizite Typdefinitionen

Bei der bisher ausschließlich verwendeten expliziten Typdefinition wird zunächst der Typname vereinbart und

dieser dann für die Deklaration von Variablen herangezogen. Manchmal genügt aber die folgende implizite Form, die an einem Beispiel eines Char-Ausschnitts gezeigt wird. Die Typbeschreibung findet hier direkt in der Variablenvereinbarung statt, ein Typname wird gar nicht erst eingeführt:

```
VAR
Index1, Index2, Index3: 'a'..'z';
```

Dies entspricht mehr oder weniger den folgenden Vereinbarungen:

```
TYPE
IndexTyp = 'a'..'z';
VAR
Index1, Index3, Index3: IndexTyp;
```

Die beiden Formen sind nicht ohne weiteres austauschbar. Die Typbeschreibung kann nicht immer für den Typnamen stehen. Wird zum Beispiel eine Variable *Index4* an anderer Stelle ebenfalls mit: *Index4 : 'a'..'z';* deklariert, so erkennt der Compiler nicht, daß es sich um den gleichen Typ wie bei *Index1, Index2, Index3* handeln soll (gleichgültig, ob die drei Variablen implizit oder explizit vereinbart wurden).

Merkzettel

Wenn getrennt deklarierte Variablen als typidentisch gelten sollen, müssen sie explizit demselben Typnamen zugeordnet sein.

In vielen Situationen ist daher die explizite Definition sinnvoller oder sogar die allein mögliche.

Wie Sie schon im letzten Kapitel an den Eingabekontrollen für Char-Variable sahen, wird eine Feinunterscheidung von zulässigen und unzulässigen Werten eines Typs nicht mit Teilbereichstypen, sondern häufig durch die Zusammenstellung eines entsprechenden SETs getroffen. Worum handelt es sich bei einem solchen „Set"?

SET-Typen

Ein „Warenkorb" oder eine Zusammenstellung von elementaren Objekten, ein SET, wird in Pascal als Wert eines strukturierten Datentyps aufgefaßt. Dazu gibt es dann auch die entsprechenden Variablen. Es sollen etwa

in einem Menü die zulässigen Eingaben 'v', 'h', 'n', 'g', 'x' sein ('V'orspeisen, 'H'auptgerichte, 'N'achspeisen, 'G'etränke und (ni)'X' oder englisch eXit, Ausgang). Mit einer Char-Variablen *Wahl* kann gefragt werden, ob:

```
Wahl IN ['v', 'h', 'n', 'g', 'x'] .
```

SET-Variable

Mit einer Variablen *ZulaessigeZeichen* können Sie dann auch schreiben:

```
Wahl IN ZulaessigeZeichen
```

Das erleichtert zum Beispiel die Eingabekontrolle außerordentlich, weil nun eine einzige Prozedur für alle Menüs reicht. Vor dem Aufruf der Prozedur muß lediglich in *ZulaessigeZeichen* der richtige Inhalt kommen:

```
ZulaessigeZeichen
     := ['v', 'h', 'n', 'g', 'x'];
```

In der Prozedur steht nur noch:

```
WHILE NOT (Wahl IN ZulaessigeZeichen) DO
  BEGIN
    read (Wahl);
                                { ... }
  END;
```

Deklaration

Möchten Sie ein SET deklarieren, schreiben Sie einfach, aus welchen Elementen das SET gebildet werden darf:

```
VAR
  ZulaessigeZeichen: SET OF Char;
```

Dies ist allerdings eine implizite Typangabe. In der zu bevorzugenden expliziten Form lautet die entsprechende Vereinbarung:

```
TYPE
  TypZeichenGruppe = SET OF Char;
VAR
  ZulaessigeZeichen : TypZeichenGruppe;
```

Der strukturierte Typ *TypZeichenGruppe* wird auch ein SET-Typ genannt (da er Zeichen in Form von SETs organisiert). In einem SET vom *TypZeichenGruppe* können Sie auch, was mit den Teilbereichstypen nicht gelingt, sämtliche Kleinbuchstaben eines (nicht englischen) Alphabets erfassen:

```
Kleinbuchstaben : TypZeichenGruppe;
```

Am Programmanfang wird der Variablen der entsprechende Inhalt zugewiesen:

```
Kleinbuchstaben := ['a'..'z', 'ä', 'ö', 'ü', 'ß'];
```

Sie sehen, daß auch bei der Auflistung der Werte in einem SET ein zusammenhängender Abschnitt eines Aufzählungstyps in abgekürzter Schreibweise aufgenommen werden darf.

SETs und ihre Elemente

Ein SET-Typ wird als strukturiert bezeichnet, weil er Werte eines einfacheren Typs in bestimmter Weise zusammenfaßt: Ein SET ist dadurch charakterisiert, daß die Reihenfolge der Nennung (und sogar die Mehrfachnennung) seiner Elemente unerheblich ist. Es zählt nur, welche Werte aus der zur Auswahl stehenden Gesamtheit – dem Elementtyp – dazu gehören und welche nicht.

Mögliche Elementtypen

Der Elementtyp, auch Grundtyp genannt, aus dessen Werten das SET gebildet wird, muß in Pascal ein Aufzählungstyp sein, dessen Werteliste einen gewissen (von der Pascal-Implementation abhängigen) Umfang nicht überschreiten darf. Diese Beschränkung rührt daher, daß intern für jedes mögliche Element eines SETs, das heißt für jeden Wert des Elementtyps, ein Bit im Speicher reserviert werden muß. Auf dieser Speicherstelle wird vermerkt, ob das betreffende Element im aktuellen SET enthalten ist oder nicht. In Turbo-Pascal wie in Quick-Pascal dürfen daher praktisch nur SETs aus Chars, aus dem Teilbereich Byte (0..255) von Integer und aus selbstdefinierten Aufzählungs- oder Teilbereichstypen gebildet werden. Ein Beispiel eines SET-Typs, dessen Elementtyp nicht Char ist: Mit *TypApfelsorte* lassen sich hervorragend diverse Apfelgruppen beschreiben. Es sei definiert:

```
TYPE
TypApfelgruppe = SET OF TypApfelsorte;

VAR
Sommeraepfel, Herbstaepfel, Winteraepfel,
Eigenzuechtungen, Neuerwerbungen, Ausgeschieden
                                { ... }
: TypApfelgruppe;

Sommeraepfel := [Klarapfel, Gravensteiner, SommerCox];
Herbstaepfel := [Boskop, Ontario, CoxOrange];
```

Merkzettel

Zu beachten ist, daß weder STRING- noch Real-Werte in einem SET vorkommen können, weil sie niemals einem Aufzählungstyp angehören.

Etwas SET-Theorie

Was heißt eigentlich das englische Wort set? Am besten fahren Sie, wenn Sie sich vom Anklang an „Geschirrset" oder „Kochlöffelset" leiten lassen. Die korrekte Übersetzung des Pascalbegriffs SET ist allerdings der unglücklich gewählte Begriff der mathematischen Menge, der anders als in der Alltagssprache nichts Quantitatives meint. Nur werden in der Mathematik die Elemente einer Menge durch geschweifte Klammern { } zusammengefaßt, die in Pascal schon für Kommentare vergeben sind.

Anders als in der mathematischen Mengenlehre (genaugenommen auch dort nur in der sogenannten „naiven" Mengenlehre) dürfen Sie in Pascal SETs nicht frei bilden: Wie Sie wissen, müssen die Elemente alle vom (selben) Elementtyp sein. Ansonsten kann vieles aus der mathematischen Mengenlehre in der „Set-Lehre" von Pascal nachvollzogen werden.

Die leere Menge

[ ] ist die „leere Menge", das leere SET, der Warenkorb ohne Inhalt. Jede Abfrage, ob irgendein Element da drin ist, führt zum Ergebnis „false". In ein Programm werden Sie natürlich nicht einfach hinschreiben

```
if Zeichen in [ ]
```

Es kann aber durchaus passieren, daß sich zum Beispiel die Variable *ZulaessigeZeichen* in der Bedingung

```
if Zeichen in ZulaessigeZeichen
```

im Zuge des Programmablaufs „entleert".

Operationen mit SETs

Wie in der Mathematik dürfen zwei Mengen vereinigt werden, in Pascal mit + geschrieben, und es kann ihr Durchschnitt gebildet werden, als Symbol dient das Multiplikationszeichen *. Die Vereinigung $A + B$ zweier SETs A, B besteht, wie der deutsche Begriff richtig vermuten läßt, aus all den Elementen, die in A oder B enthalten sind. Zum Durchschnitt $A * B$ gehören dagegen nur die Elemente, die sowohl in A wie auch in B liegen. Auch die Differenzmenge gibt es in Pascal, die wie in der Mathematik mit dem gewöhnlichen Minuszeichen ausgedrückt wird: Zu $A - B$ gehören die Elemente, die in A, aber nicht in B liegen.

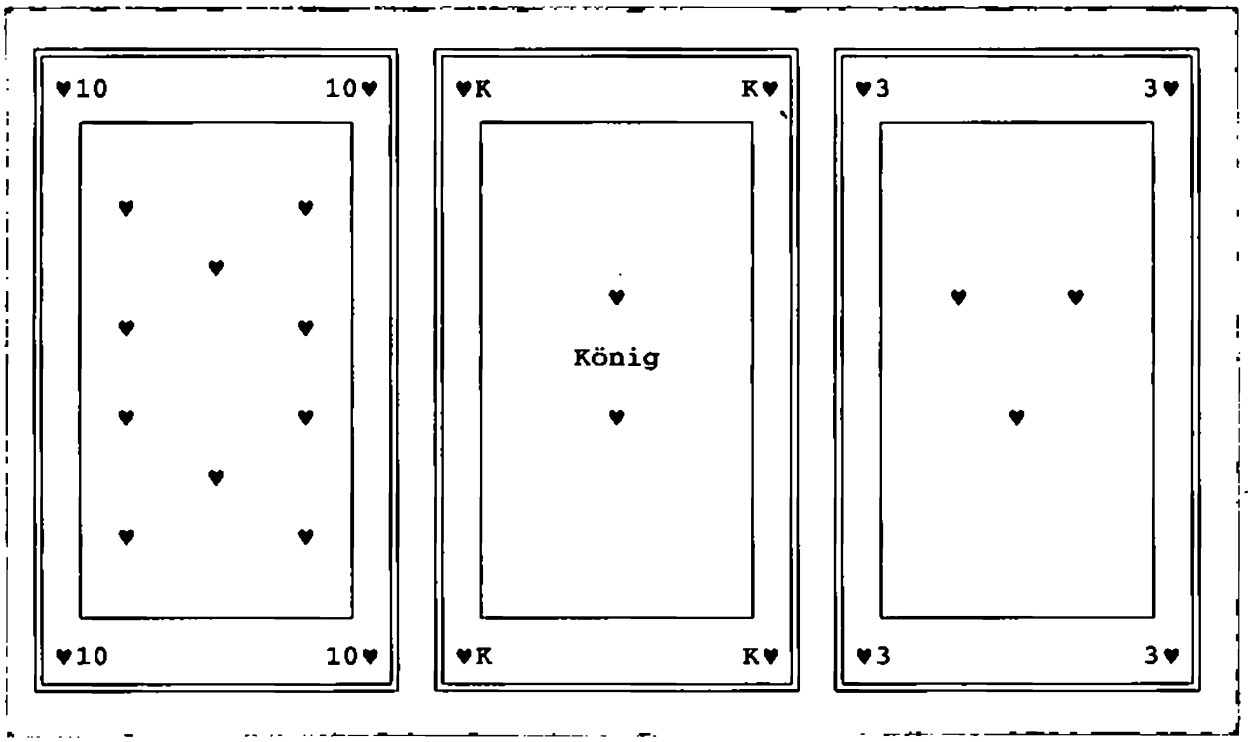

Abbildung 12-3: Einfache Beispielgestaltung für Spielkarten

Zwei SETs (des gleichen Elementtyps) können mit = auf Gleichheit befragt werden. Außerdem können Sie (mit Größer- bzw. Kleinerzeichen) feststellen, ob das eine im anderen (als ein „Teil", eine Teilmenge) enthalten ist. Im folgenden Beispielprogramm stellen Sie mit Hilfe der ziemlich theoretisch wirkenden Mengenoperationen auf sehr anschauliche Weise ein *KartenSet* zusammen.

Vergleiche

Beispiel: Karten ziehen

Mit dem folgenden Programm können Sie ein Blatt Karten aus einem Spiel (der Kürze halber nur Herz) „ziehen" und, wenn Sie wollen, es sich leicht auch ausdrucken. Der vollständige Kartensatz erscheint als *Typkarte*. Die noch im *Stapel* auf dem Tisch liegenden Karten sind ebenso wie die Karten auf der Hand als Variable des Typs SET OF *TypKarte* modelliert. Die verzwickteste Stelle des Programms ist wohl das zufällige Ziehen einer Karte aus dem Stapel. Es wird simuliert, indem eine zufällige Zahl *X* bestimmt wird, die zwischen 1 und der Anzahl der Karten auf dem Stapel liegt. Anschließend müssen Sie die X-te Karte im Reststapel (der aber als geordnet angenommen werden kann) suchen. Die Prozeduren, die die Karten zeichnen sollen, sind nur als einfache Dummy-Prozeduren aufgeführt. Wenn Sie mögen, gestalten Sie sich einen möglichst schönen Kartensatz mit den zur Ver-

fügung stehenden graphischen Möglichkeiten. Relativ leicht geht es noch bei den Zahlenkarten (Abb. 12-3 auf Seite 199):

```
PROGRAM KartenSpiel;
USES            {Entfällt in Turbo 3.x}
  crt;

CONST
  Verzoegerungszeit = 500;
  Anfangszeile  = 12;
  Anfangsspalte = 41;

TYPE
  TypKarte      = (As, Sieben, Acht, Neun, Zehn,
                   Bube, Dame, Koenig);
  TypKartenSet = SET OF TypKarte;
  Cardinal      = 0 .. MaxInt;

VAR
  AnzahlAufStapel           : Cardinal;
  Zahl, Gesamtzahl          : Cardinal;
  Karte                     : TypKarte;
  Stapel, Hand              : TypKartenSet;
  Zeichen                   : Char;

PROCEDURE AsMalen;
BEGIN
  WriteLn;
                            {Rahmen aus ASCII-Code: 219}
  WriteLn ('▌▌▌▌▌▌▌▌▌');
  WriteLn ('▌▌▌',#3,'As▌▌▌');
  WriteLn ('▌▌▌▌▌▌▌▌▌');
END;
PROCEDURE SiebenMalen;
BEGIN
  WriteLn ('▌▌▌▌▌▌▌▌▌');
  WriteLn ('▌▌▌',#3,'7▌▌▌');
  WriteLn ('▌▌▌▌▌▌▌▌▌');
END;
PROCEDURE AchtMalen;
BEGIN
  WriteLn ('▌▌▌▌▌▌▌▌▌');
  WriteLn ('▌▌▌',#3,'8▌▌▌');
  WriteLn ('▌▌▌▌▌▌▌▌▌');
END;
PROCEDURE NeunMalen;
```

```
BEGIN
  WriteLn ('▮▮▮▮▮▮▮▮▮');
  WriteLn ('▮▮▮',#3,'9▮▮▮');
  WriteLn ('▮▮▮▮▮▮▮▮▮');
END;
PROCEDURE ZehnMalen;
BEGIN
  WriteLn ('▮▮▮▮▮▮▮▮▮');
  WriteLn ('▮▮▮',#3,'10▮▮▮');
  WriteLn ('▮▮▮▮▮▮▮▮▮');
END;
PROCEDURE BubeMalen;
BEGIN
  WriteLn ('▮▮▮▮▮▮▮▮▮');
  WriteLn ('▮▮▮',#3,'Bu▮▮▮');
  WriteLn ('▮▮▮▮▮▮▮▮▮');
END;
PROCEDURE DameMalen;
BEGIN
  WriteLn ('▮▮▮▮▮▮▮▮▮');
  WriteLn ('▮▮▮',#3,'Da▮▮▮');
  WriteLn ('▮▮▮▮▮▮▮▮▮');
END;
PROCEDURE KoenigMalen;
BEGIN
  WriteLn ('▮▮▮▮▮▮▮▮▮');
  WriteLn ('▮▮▮',#3,'Kö▮▮▮');
  WriteLn ('▮▮▮▮▮▮▮▮▮');
END;
PROCEDURE Anzeigen;
BEGIN
  WriteLn;
  CASE Karte OF
    As     : AsMalen;
    Sieben : SiebenMalen;
    Acht   : AchtMalen;
    Neun   : NeunMalen;
    Zehn   : ZehnMalen;
    Bube   : BubeMalen;
    Dame   : DameMalen;
    Koenig : KoenigMalen;
  END;
END;
```

```
PROCEDURE Ziehen;
VAR
  i, X : Cardinal;
BEGIN
  IF AnzahlAufStapel > 0
    THEN
      BEGIN
        X := Random (AnzahlAufStapel) + 1;

             {Ergibt zufällige Zahl zwischen 1 und
             AnzahlAufStapel. Jetzt suche die X-te Karte
             der auf dem Stapel verbliebenen Karten:}

        Karte := As;       {Anfangswert}
        WHILE NOT (Karte IN Stapel) DO
          Karte := succ (Karte);

             {Karten, die nicht mehr auf dem Tisch liegen, werden
             übersprungen. Wenn X größer als 1 ist, weitergehen...}

        FOR i:= 2 TO X DO
          BEGIN
            Karte := succ (Karte);
                     {Zur nächsten Karte gehen und wieder:}
            WHILE NOT (Karte IN Stapel) DO
              Karte := succ (Karte);
                     {Karten überspringen, die nicht mehr im
                     Stapel sind }
          END;
                     {Mittels Subtraktion von SETs
                     die gezogene Karte aus dem Stapel entfernen }
        Stapel := Stapel - [Karte];

                    {Und auch die Anzahl um 1 verringern }
        AnzahlAufStapel := AnzahlAufStapel - 1;
      END
    ELSE
      Write ('Keine Karte mehr da.');
END;

PROCEDURE BlattZeigen;
BEGIN
  Clrscr;
  FOR Karte := As TO Koenig DO
    IF Karte IN Hand THEN Anzeigen;
END;

PROCEDURE ProgrammErklaerung;
VAR
  i : Cardinal;
```

```
PROCEDURE Copyright;
  BEGIN
    WriteLn;
    WriteLn (' Verfasser:  ....  ');
    WriteLn (' (C)         19..  ');
    WriteLn;
  END;

BEGIN
  Write   ('Das Programm simuliert das Ziehen ');
  WriteLn ('eines Blattes aus den ',#3,'-Karten.');
  Copyright;
  FOR i := 1 TO 80 DO
    Write ('_');
  WriteLn;
  WriteLn ('Wieviel Karten sollen gezogen werden? ');
  WriteLn ('Geben Sie bei der einfachen Beispielgestaltung');
  Write   ('eine Zahl zwischen 1 und 8 ein:  ');
  ReadLn  (Gesamtzahl);
  WriteLn;
  WriteLn ('Zum Fortsetzen beliebige Taste drücken');
  Zeichen := ReadKey;     {Oder: Read (Kbd, Zeichen);}
  ClrScr;
END;

BEGIN (* von Hauptprogramm *)
  Randomize;
  ClrScr;
  ProgrammErklaerung;
  AnzahlAufStapel := ord (Koenig) + 1;
  Stapel   := [As .. Koenig];
  Hand     := [];
  FOR Zahl := 1 TO Gesamtzahl DO
    BEGIN
      GoToXY  (Anfangsspalte, Anfangszeile);
      Write   ('Ziehen! ');
      Zeichen := ReadKey; {Oder: Read (Kbd, Zeichen);}
      ClrScr;
      Ziehen;
      Hand    := Hand + [Karte];
      GoToXY (1, Anfangszeile);
      delay  (Verzoegerungszeit);
      Anzeigen;
      Zeichen := ReadKey; {Oder: Read (Kbd, Zeichen);}
      ClrScr;
    END;
```

```
  Write ('Das Blatt zeigen ');
  Zeichen := ReadKey;      {Oder: Read (Kbd, Zeichen);}
  BlattZeigen;
  Zeichen := ReadKey;      {Oder: Read (Kbd, Zeichen);}
END. (* von Hauptprogramm *)
```

Gezogene Karten (zum Beispiel von mehreren Spielern oder Spielerinnen) werden in ihrem Rang verglichen mit Abfragen wie:

```
Karte1 < Karte2
```

Sie können den Wert der Karten verändern, indem Sie zum Beispiel *As* in *TypKarte* hinter den *Koenig* stellen oder die *Dame* an den Platz mit dem höchsten Rang. Im Prinzip lassen sich auf diese Weise Kartenspiele programmieren. Der Aufwand wird, wie bei fast allen Programmierunternehmungen, aber schnell unübersehbar.

Erst das übernächste Kapitel wird mit der Pascal-Typologie fortfahren. Im nächsten Kapitel stellen die Typen nur das notwendige Hilfsmittel dar, um endlich alle Möglichkeiten auszuschöpfen, welche die Prozeduren in Pascal bieten.

Aufgaben

1. Ergänzen Sie *KartenSpiel* um eine Prozedur, die am Schluß noch den Punktwert des Blattes auf der Hand anzeigt (mit einer üblichen Bewertung der Karten).

2. Entwerfen Sie für die Apfelzucht eine Prozedur, mit der ein *Angebot* zusammengestellt wird. Bei jeder der lieferbaren Sorten (ein Teil-SET mit Elementen von *TypApfelsorte*) fragt die Prozedur, ob sie in das Angebots-SET aufgenommen werden soll. Am Schluß wird zur Kontrolle nochmal der gesamte „Korb" angezeigt.

3. Nutzen Sie die Möglichkeiten der SET-Variablen, um in ältere Programme Eingabekontrollen einzubauen.

13 Prozeduren und Funktionen mit Parametern

Die Einführung von Parametern – Die Unterscheidung lokaler und globaler Größen – Werte- und Variablenparameter – Modularisierung – Prozeduren für den täglichen Gebrauch – Vier weitere nützliche Bausteine – UNITs – Funktionen – Zur geschachtelten Verwendung von Prozeduren – Rekursion – Aufgaben

Was geht ein, was kommt heraus

Was geht ein, was kommt heraus – wird diese Frage beim Entwurf von Prozeduren konsequent gestellt und die Antwort in die Definition und den Aufruf der Prozeduren einbezogen, entstehen Prozeduren mit Parametern. Im Grunde sind viele Pascal-Elemente, wie Write und Read, wie die Funktionen Length, Copy und andere, wie Assign, Rewrite, Reset und Close, wie ord, pred, succ, ja sogar die mathematischen Operationen nichts anderes als Prozeduren, Hilfsprogramme, die für bestimmte Arten (Typen) von Größen eine bestimmte Aufgabe erledigen. Mit welchem Wert des festgelegten Typs (beziehungsweise welcher Variablen dieses Typs) diese vordefinierten Prozeduren arbeiten, wird in der Regel in Klammern hinter das Pascal-Wort geschrieben. Genau diese Objekte in Klammern heißen Parameter (eigentlich die „Bei-Maße") der betreffenden Prozeduren.

Die Einführung von Parametern

Das Beispiel Karten ziehen

Betrachten Sie das Programm *KartenSpiel* aus dem letzten Kapitel. Nehmen Sie an, daß Sie nach dem Ziehen der Karten nicht nur einen Blick in das Blatt auf der *Hand* werfen, sondern auch den *Stapel* auf dem fiktiven Tisch kontrollieren wollen. Wie muß das Programm ergänzt werden? Die Prozedur *BlattZeigen* läßt Sie nur in die *Hand* spicken. Sie könnten eine entsprechende Prozedur für den Stapel schreiben, indem Sie die Variable *Hand* durch *Stapel* ersetzen und sich einen neuen Namen für die Prozedur ausdenken.

```
PROCEDURE StapelZeigen;
BEGIN
  ClrScr;
  FOR Karte := As TO Koenig DO
    IF Karte IN Stapel THEN Anzeigen;
END;
```

Was aber ist, wenn noch weitere Personen mitspielen, in deren Karten „gelugt" werden soll? Muß für jede *Hand* eine eigene Prozedur geschaffen werden? Mit der Parametrisierung der ursprünglichen Prozedur läßt sich dieser unnötige Aufwand vermeiden. Wenn ein oder mehrere zusätzliche Parameter eingeführt werden – eine alltägliche Aufgabe in der Programmierpraxis – sind der Prozedurkopf und jeder Aufruf der Prozedur zu ändern:

1. In der Definition der Prozedur wird der Prozedurkopf um den (oder die) formalen Parameter samt Typnamen ergänzt. Vorsicht: Die Typbeschreibung (implizite Typdeklaration) genügt nicht, es muß tatsächlich ein Typname angegeben sein!

2. Bei jedem Aufruf ist ein aktueller Parameter (meist eine Variable) anzugeben, der vom selben Typ sein muß wie der entsprechende formale Parameter in der Deklaration. Der Gebrauch des Wortes „aktuell" meint in diesem Zusammenhang die Parameter, mit denen tatsächlich gearbeitet wird, das heißt, die für einen bestimmten Aufruf das effektive „Beimaß" der Prozedur darstellen. Sie werden demgemäß auch Aufruf- oder Aktualparameter genannt.

In der Prozedurdeklaration des Beispiels wird als formaler Parameter etwa *KartenGruppe* vom *TypKartenSet* festgelegt:

```
PROCEDURE BlattZeigen (Kartengruppe : TypKartenSet);
BEGIN
  ClrScr;
  FOR Karte := As TO Koenig DO
    IF Karte IN Kartengruppe THEN Anzeigen;
END;
```

Möchten Sie jetzt tatsächlich in die *Hand* schauen, wird die Prozedur mit *Hand* als Beimaß aufgerufen:

```
BlattZeigen (Hand);
```

Ganz analog decken Sie den *Stapel* auf:

```
BlattZeigen (Stapel);
```

Mittels eines entsprechenden Aufrufs gibt also die eine Prozedur *BlattZeigen* über jede Variable des *TypKartenSet* Auskunft.

Funktionsweise

Das Pascal-Wort *Kartengruppe*, das als formaler Parameter dient, ist eigentlich nur ein Hilfsmittel zur Beschreibung des Ablaufschemas der Prozedur. Beim Aufruf wird der jeweilige Aktualparameter praktisch für *Kartengruppe* eingesetzt. All das, was im Prozedurinneren für den Formalparameter beschrieben war, geschieht mit dem Aufrufparameter.

Änderungsfreundlichkeit dank parametrisierter Prozeduren

Ein wesentliches Qualitätskriterium für Programme ist deren Änderungsfreundlichkeit: Mit welchem Aufwand läßt sich das Programm anpassen, wenn die Anforderungen geringfügig variieren. Wie Sie beobachten, gestattet die parametrisierte Prozedur, ohne Schwierigkeiten einen Blick in zusätzliche Kartengruppen zu werfen.

Es wäre übrigens nicht nötig gewesen, sich ein eigenes Pascal-Wort für den formalen Parameter auszudenken. Man hätte *Hand* als Bezeichnung nehmen und dennoch nachher die Prozedur für *Stapel* aufrufen können.

```
BlattZeigen (Stapel);
```

steht formal nicht im Widerspruch zu folgender Deklaration, die völlig gleichwertig zu der mit *Kartengruppe* ist:

```
PROCEDURE BlattZeigen (Hand : TypKartenSet);
BEGIN
  FOR Karte := As TO Koenig DO
    IF Karte IN Hand THEN Anzeigen;
END;
```

Vorsicht Falle

Achten Sie darauf, nicht gleich in eine der übleren Fallen von Pascal-Programmen zu stolpern. Nimmt man versehentlich als Parameternamen *Kartengruppe*, arbeitet aber im Prozedurinneren weiter mit *Hand*, läßt sich die Prozedur mit *Stapel* aufrufen, aber zeigt die Karten aus der *Hand*.

Weitere Parametrisierungen

Die Möglichkeit, einen Variablennamen des Hauptprogramms auch als Parameternamen zu verwenden, sei jetzt genutzt, um bequem *Anzeigen* zu parametrisieren. Ein neuer Prozedurkopf lautet:

```
PROCEDURE Anzeigen (Karte : TypKarte);
```

Sonst müssen Sie an der Deklaration nichts ändern. Wenn Sie das Programm entsprechend korrigieren, müssen Sie nur daran denken, daß ab jetzt auch im Aufruf ein Parameter stehen muß. Nach Herzenslust können Sie im Programm nun zusätzliche Variablen vom *TypKarte* einbauen und sich jede anzeigen lassen, etwa:

```
HoechsteKarte, NiedrigsterWert, TrumpfAs : TypKarte;
```

Die Unterscheidung lokaler und globaler Größen

Ob eine Größe als Parameter der Prozedur auftreten soll, hat viel mit der Unterscheidung lokaler und globaler Variablen zu tun. Eine Variable ist lokal, ortszugehörig, in der Prozedur, in der sie deklariert wurde. Global heißen Variable, die in einer übergeordneten Prozedur beziehungsweise dem Hauptprogramm vereinbart wurden. Ganz entsprechend kann auch von lokalen und globalen Konstanten, Typen und Prozeduren gesprochen werden. Lokal und global sind strenggenommen relative Begriffe: Die Variable i der Prozedur *Programmerklaerung* ist lokal im Verhältnis zum Hauptprogramm, aber global gegenüber der Subprozedur *Copyright*.

Als Grundsatz kann gelten: Alle Objekte (Variable, Typen, Konstanten, aber auch Prozeduren, auf die ebenfalls die Begriffe lokal und global anwendbar sind) sollen an dem Ort angesiedelt werden, an dem sie ihren Sinn und ihre Funktion haben.

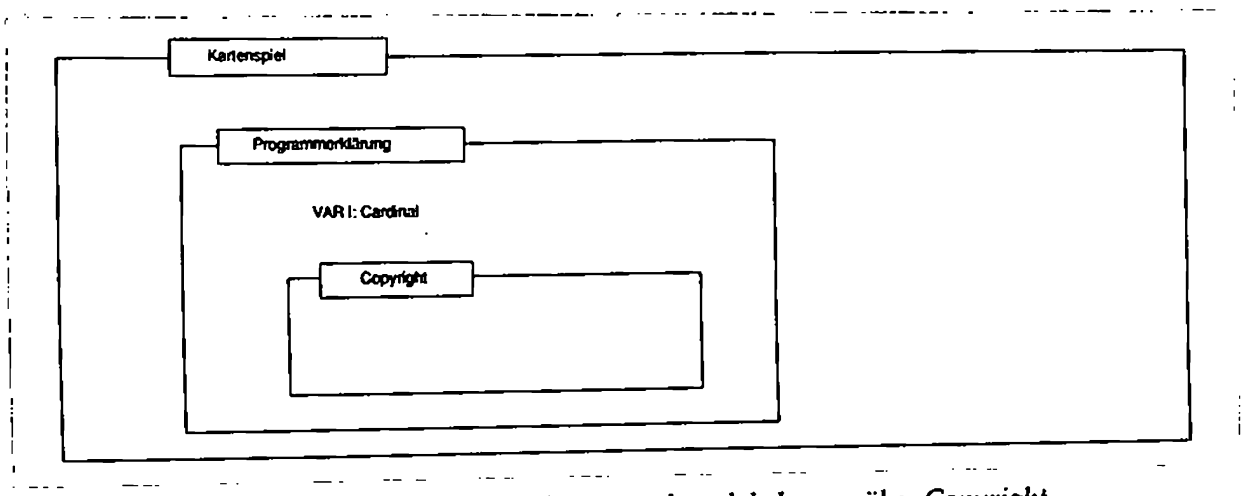

Abbildung 13-1: i ist lokal in Programmerklaerung, aber global gegenüber Copyright.

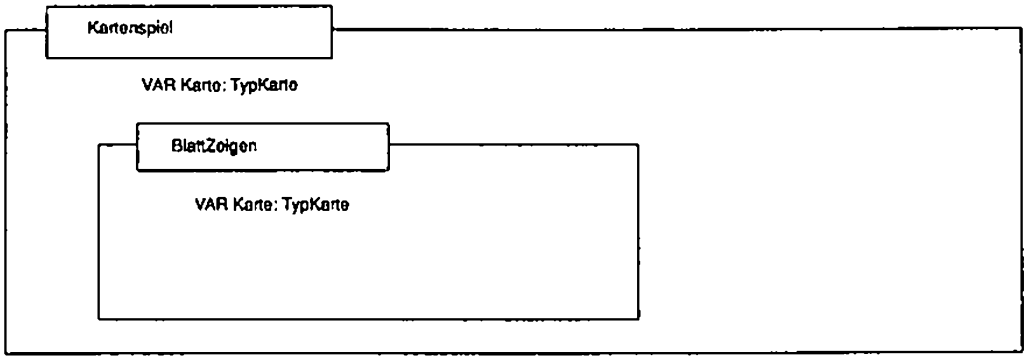

Abbildung 13-2: Karte „innen" verdeckt Karte „außen", solange die Prozedur arbeitet.

Zum Beispiel hat die Variable *Karte* in der Prozedur *BlattZeigen* eine vollständig ortsbezogene Bedeutung. Sie dient nur als Hilfsgröße, um alle Werte von *As* bis *Koenig* abzuklappern. Was in der Prozedur mit ihr geschieht, soll keinerlei Verbindung mit Vorgängen außerhalb haben. Deshalb erhält *Karte* besser eine lokale Deklaration (genauso wie schon *i*, *X* in *Ziehen* und *i* in *Programmerklaerung*):

```
PROCEDURE BlattZeigen (Kartengruppe : TypKartenSet);
VAR
  Karte : TypKarte;
BEGIN
  ClrScr;
  FOR Karte := As TO Koenig DO
    IF Karte IN Kartengruppe THEN Anzeigen;
END;
```

„Verdecken" eines Pascal-Wortes

Es macht gar nichts, daß außerdem *Karte* als globale Variable existiert. Wenn ein global definierter Pascal-Name (gleichgültig ob für Variable, Typen, Konstanten, oder Prozeduren) lokal neu vereinbart wird, so wird davon gesprochen, daß die globale Bedeutung innerhalb der Prozedur verdeckt oder ausgeblendet ist. Solange die Prozedur arbeitet, haben die örtlichen Namen Vorrang. Nachher verschwinden diese wieder, und die globalen Größen kommen zum Vorschein.

Seiteneffekte

Einer der unangenehmsten Fehler, der in Pascal-Programmen auftreten kann (oben erwähnte „Fälle"), kommt zustande, wenn in einer Prozedur versehentlich mit einer globalen Variablen hantiert wird, weil die Neudeklaration oder eine Umbenennung vergessen wurde. In *BlattZeigen* zum Beispiel trägt *Karte* am Schluß stets

den Wert *Koenig*. In der ursprünglichen Programmfassung stört das nur deshalb nicht, weil die Prozedur am Programmschluß aufgerufen wird. Würde man dem Spieler oder der Spielerin erlauben, unmittelbar nach dem Ziehen ins Blatt zu schauen, vergäße aber, *Karte* lokal zu definieren, so wäre auf einmal ein böser Taschenspielertrick installiert. Scheinbar würde immer der *Koenig* gezogen, da er den Endwert der FOR-Schleife darstellt. Wenn das Unterprogramm mit den Objekten der übergeordneten Prozedur arbeitet und oft schier unerklärliche Effekte bewirkt, wird von einem Seiteneffekt des Unterprogramms gesprochen.

Werte- und Variablenparameter

In folgender Abwandlung kann *KartenSpiel* schon recht unterhaltsam sein. Zwei Personen sollen abwechselnd je eine Karte ziehen, wer jeweils die höhere findet, darf beide behalten. Gewonnen hat, wer am Schluß die höchste Punktzahl auf der Hand hält. Das Programm benötigt, um zwei Karten vergleichen zu können, zwei Variable, etwa *KarteA* (der ersten Person zugehörig) und *KarteB* (von der anderen Person gezogen), beide vom *TypKarte*. Die Abfrage lautet dann:

```
IF KarteA > KarteB
```

Damit die beiden Spieler die Karten aufnehmen können, werden noch zwei Variablen *HandA* und *HandB* vom *TypKartenSet* benötigt. Die Prozedur *Ziehen* bringt im ursprünglichen Programm einen zufälligen Wert auf *Karte*. *Ziehen* läßt sich parametrisieren, um beliebige Variablen vom *TypKarte* zu besetzen. Jedoch unterscheidet sich die Aufgabe der Parameter in den Prozeduren *Anzeigen* oder *BlattZeigen* auf der einen, *Ziehen* auf der anderen Seite in einem wichtigen Punkt. Die oben besprochenen „schlukken" vorher vorhandene Werte und verarbeiten sie im Inneren. *Ziehen* dagegen gibt im Inneren einer Größe einen Wert, der nachher außen gebraucht wird. In der ersten Situation wird von einem Werteparameter gesprochen, in der zweiten handelt es sich um einen sogenannten Variablenparameter. In der Deklaration eines Parameters der zweiten Art steht zusätzlich VAR vor dem Parameternamen:

```
PROCEDURE Ziehen (VAR Karte : TypKarte);
VAR
  i, X : Cardinal;
BEGIN
                                { ... }
END;
```

Der Aufruf erfolgt ohne besondere Kennzeichen:

```
Ziehen (KarteA);
Ziehen (KarteB);
```

Ein Werteparameter trägt eigentlich nur den Wert einer Variablen oder eines Ausdrucks in das Prozedurinnere. Dort ist eine (systeminterne) lokale Variable reserviert, auf die eine Kopie des Wertes gebracht wird. Mit dieser Kopie kann nach Herzenslust gearbeitet werden. Was auch immer mit ihr geschieht, ist nach Verlassen der Prozedur wieder vergessen. Ein Fehler innerhalb der Prozedur kann im Hauptprogramm keinen Schaden anrichten. Beim Variablenparameter dagegen arbeitet die Prozedur tatsächlich mit der globalen Variablen, die im Aufruf steht. Mitsamt der Variablen „schlüpft" ihr Wert nach innen. Auch wandert ein eventueller neuer Wert, der ihr dort verliehen wird, mit in den äußeren Programmteil.

Merkzettel

Werteparameter sind Einbahnstraßen und dienen nur dem Datenimport (was geht ein), Variablenparameter exportieren auch Daten (was kommt heraus).

Ausnahme

Bei sehr großen Datenobjekten wird allerdings in Anbetracht des Speicherplatzes, den die lokale Datenkopie bei Werteparametern erfordert, einem Variablenparame-

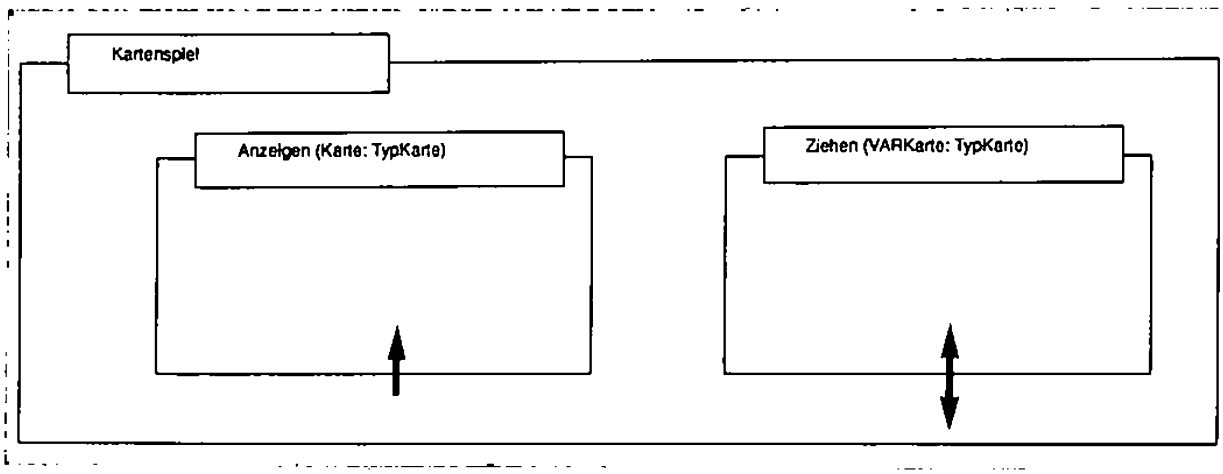

Abbildung 13-3: Wert- und Variablenparameter

ter auch dann der Vorzug gegeben, wenn die Daten nur importiert werden sollen.

Zwei, drei, viele Parameter

Was ist eigentlich mit den beiden Objekten *Stapel* und *AnzahlAufStapel* in der Prozedur *Ziehen*? Sollten sie nicht auch in Parameter umgewandelt werden? Wenn, dann jedenfalls in Variablenparameter, denn es ist eine für den weiteren Programmablauf global interessante Angelegenheit, welche Karten im Stapel verbleiben. Es ist ohne weiteres möglich, einer Prozedur auch zwei oder mehr Parameter zu geben – inwiefern dies bei *Ziehen* Sinn macht, wird später diskutiert. In der Definition sind die Beschreibungen der einzelnen Parameter durch Semikolon zu trennen, beim Aufruf steht zwischen den Aktualparametern das Komma. Der Prozedurkopf von *Ziehen* könnte also wie folgt ergänzt werden:

```
PROCEDURE Ziehen
  (VAR Karte: TypKarte;
   VAR Stapel: TypKartenSet;
   VAR AnzahlAufStapel: Cardinal);
```

Bei jedem Aufruf müssen dann ebenfalls drei Parameter stehen, der erste eine Variable vom *TypKarte*, der zweite vom *TypKartenSet*, der dritte *Cardinal*:

```
Ziehen (KarteA, Stapel, AnzahlAufStapel);
```

Das Pascal-System ordnet Formal- und Aktualparameter einander entsprechend ihrer Reihenfolge zu. Dabei dürfen Werte- und Variablenparameter beliebig gemischt vorkommen und auch mehrere Parameter desselben Typs vereinbart werden. Vielleicht ist es nützlich, an dieser Stelle das Syntaxdiagramm für die Deklaration des Prozedurkopfes (und getrennt die Parameterliste) aufzuzeichnen (siehe Abb. 13-4 und 13-5).

Bemerken Sie in Abb. 13-5 den Schlenker, der hinter *Parametername* abzweigt und über ein Komma erneut auf *Parametername* führt? Er besagt, daß mehrere Parameter desselben Typs (wenn Sie entweder sämtlich Variablen- oder sämtlich Werteparameter sind) auch gemeinsam aufgeführt werden dürfen. Als Beispiel sei der Kopf einer Prozedur *Spielrunde* aufgeschrieben. Die Prozedur soll die vorher gezogenen Werte *KarteA* und *KarteB* vergleichen und je nach Ausgang die beiden Karten zu *HandA* oder *HandB* hinzufügen:

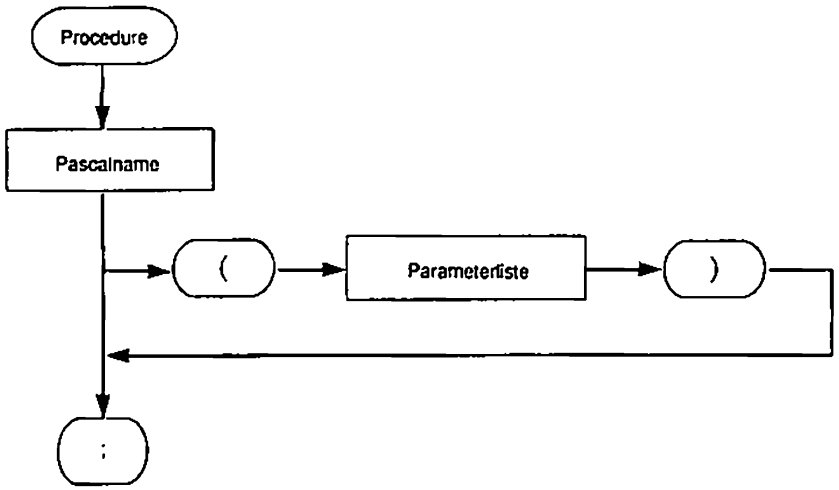

Abbildung 13-4: Syntaxdiagramm Prozedurkopf

```
PROCEDURE Spielrunde
  (KarteA, KarteB      : TypKarte;
   VAR HandA, HandB    : TypKartenSet);
BEGIN
  IF KarteA > KarteB
    THEN
      HandA := HandA  +  [KarteA, KarteB]
    ELSE
        {KarteA = KarteB ist unmöglich, es kann daher von
        KarteB < KarteA ausgegangen werden.}

      HandB := HandB + [KarteA, KarteB];
END;
```

Modularisierung

Wie Sie sehen, ist es möglich, eine Prozedur komplett zu parametrisieren. Jeder Werteaustausch mit dem übergeordneten Programm vollzieht sich über die ausgewiesene Schnittstelle, das heißt die Parameter. Ansonsten arbeitet die Prozedur ausschließlich mit lokalen Variablen, die in ihr deklariert sind. Die Prozedur darf sich sozusagen ihr eigenes kleines Reich schaffen, in das ihr niemand hineinredet. Nur die Parameter erlauben dann einen genau geregelten Datenverkehr mit ihrer Außenwelt. Die Idealform einer selbständigen Programmeinheit, die ausschließlich über definierte Schnittstellen, welche den Rahmen der möglichen Informationsflüsse festlegen, mit

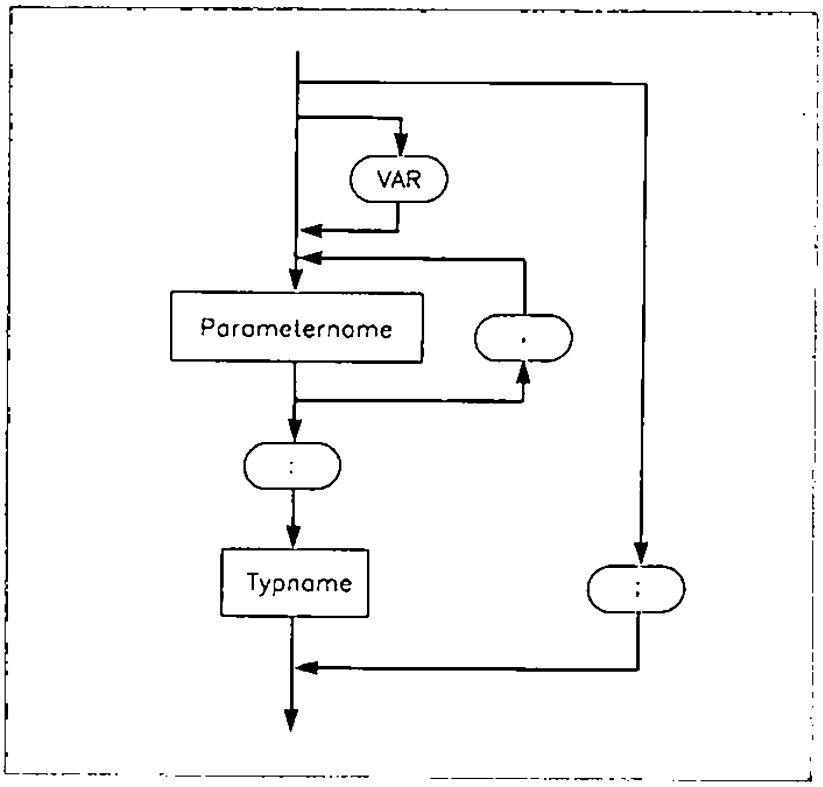

Abbildung 13-5: Syntaxdiagramm Parameterliste

anderen Einheiten in Verbindung tritt, wird Modul genannt.

Modularisierung heißt das Prinzip der Aufteilung einer großen Aufgabe in strikt voneinander getrennte Teilprobleme. Sie bildet den wesentlichen Ansatz des „Software Engineering", der Lehre von der ingenieurmäßigen Herstellung von Software. Die damit verbundenen Erwartungen gründen in der Annahme, daß in einem großen Projekt kleine Bausteine sicher beherrschbar sind und so gefertigt werden, daß sie unabhängig von ihrer Umgebung funktionieren. Daraus wird abgeleitet, daß nachher das Ganze im gewünschten Sinn arbeitet.

Aber auch wenn der Ansatz die Schwierigkeiten des Programmierens nicht aufheben kann, ist die modulare Herangehensweise beim Entwerfen und Testen sehr hilfreich. Ein Modul kann geschrieben werden, ohne daß das umgebende Programm näher bekannt sein muß. Formal ist er ohne weiteres ersetzbar durch jede Einheit, die über eine identische Schnittstelle verfügt. Die Fortentwicklungen von Pascal, unter anderem Modula2 und Oberon, verfügen über ein ausgeführtes Modulkonzept. Aber auch UCSD-, Turbo- und Quick-Pascal haben mit den Units (vergleiche unten) so etwas wie Moduln in die

Sprache aufgenommen. Im herkömmlichen Pascal sollte man das Prinzip der Modularisierung als Leitfaden für das Schreiben parametrisierter Prozeduren betrachten.

In welchem Ausmaß dieses Prinzip befolgt werden soll, ist allerdings eine Sache der Programmierphilosophie und auch des dafür zu treibenden Aufwandes. Es kann durchaus sinnvoll sein, mit globalen Variablen statt mit Parametern zu arbeiten. Der reinen Lehre der Modularisierung entspricht zum Beispiel die Prozedur *Ziehen*. Praktisch unumgänglich und notwendig ist die Parametrisierung von *Stapel* und *AnzahlAufStapel* hier jedoch nur, wenn es tatsächlich verschiedene *Stapel* zum Kartenziehen gibt (oder mit einer entsprechenden Programmerweiterung zu rechnen ist). Wenn dagegen klar ist, daß sich alle Prozedurhandlungen stets mit ein- und demselben Stoß Karten beschäftigen, können Sie auch die Auffassung vertreten, daß eine derartige globale Größe grundsätzlich allen Prozeduren zur Verfügung steht, ohne weitere formale Kontrolle.

Top-down

Die Frage der Parametrisierung hängt eng mit den schon in Kapitel 3 erwähnten „top down"- und „bottom up"-Programmiertechniken zusammen. Nach dem Top-down-Schema ist die Gesamtaufgabe grob in Teilprobleme, Prozeduren zu gliedern. Die Teile sind weiter zu analysieren, von ihnen die Teile wieder zu zerlegen, bis man schließlich bei den elementaren Befehlen der Programmiersprache angekommen ist.

Bottom-up

Bottom-up kann so aufgefaßt werden, daß derselbe Weg, nur diesmal vom Boden nach oben, gegangen wird. Zielorientiert werden der Sprache Elemente, Prozeduren

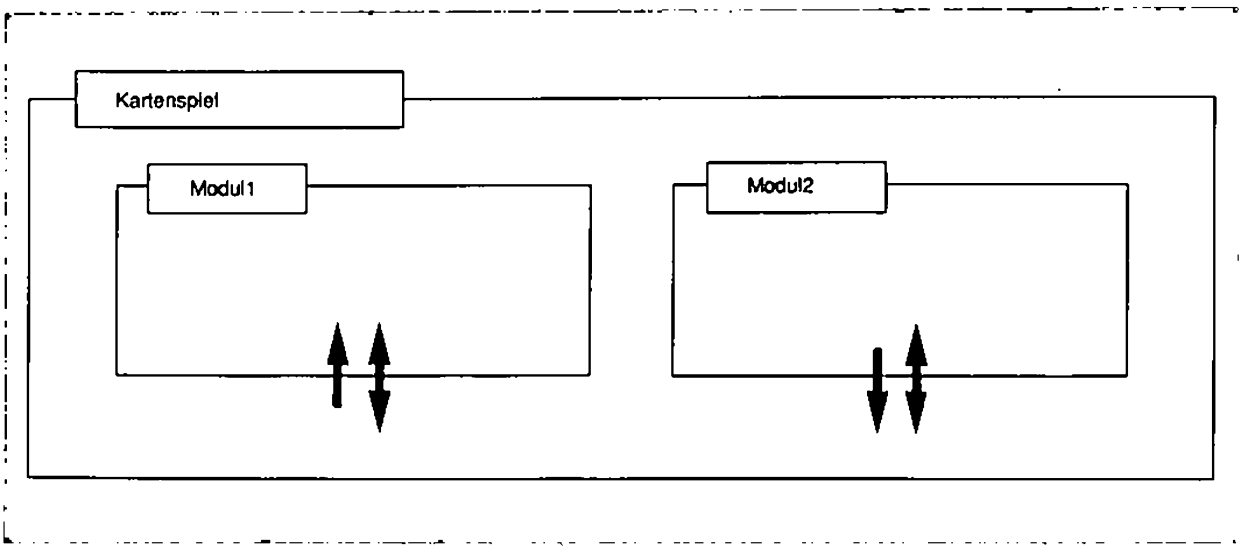

Abbildung 13-6: Moduln mit Schnittstellen als Bausteine eines Gesamtprojekts

hinzugefügt, bis sich die große Aufgabe unmittelbar lösen läßt. Der Akzent kann aber auch ganz auf diese Erweiterung der Sprachmittel gesetzt werden. Man hat weniger ein bestimmtes Ziel (Programm) im Auge, sondern bemüht sich, der Sprache universell verwendbare und leistungsfähige Elemente hinzuzufügen. Nicht so sehr die hierarchische Struktur des Programms, sondern die Elementaranweisungen E1, ... sind entscheidend (Abb. 13-7).

Module und parametrisierte Prozeduren können als universelle Sprachelemente für die unteren Ebenen diverser Programme gedeutet werden. Weil sie sozusagen autark sind, lassen sie sich in verschiedene Zusammenhänge einfügen. Von ihnen wird eine bestimmte, formale Funktion verrichtet, die diversen Zwecken dienen kann.

Prozeduren für den täglichen Gebrauch

Es gibt Funktionen, die in ähnlicher Form immer wieder benötigt werden. Sie können sich eine Sammlung solcher Hilfsmittel als eine Art Baukasten in einer Datei zusammenstellen, die Sie bei Bedarf in ein Programm hineinkopieren. Solche fertigen Werkzeuge (Tools) werden auch von vielen Firmen für unterschiedliche Zwecke und Pascal-Versionen (und andere Sprachen) angeboten.

Include-Compileroption

Um sparsamer zu wirtschaften und auch den Editor nicht unnötig zu beladen, können Sie die Include-Option des Compilers verwenden (include, „einschließen, hineinnehmen"). Dazu schreiben Sie den Dateinamen der Prozedurensammlung (etwa BAUKAST.INC) wie einen Kommentar in den Programmtext, wobei unmittelbar hinter der öffnenden Kommentarklammer $I stehen muß:

```
{$I BAUKAST.INC}
```

Nun wird der gesamte Text der Include-Datei ins Compilieren einbezogen, als stünde er selbst in der Hauptdatei (vorausgesetzt natürlich, BAUKAST.INC befindet sich auf der richtigen Diskette und im richtigen Unterverzeichnis). Mit dieser Methode können auch Programme angegangen werden, die das Fassungsvermögen des Editors übersteigen.

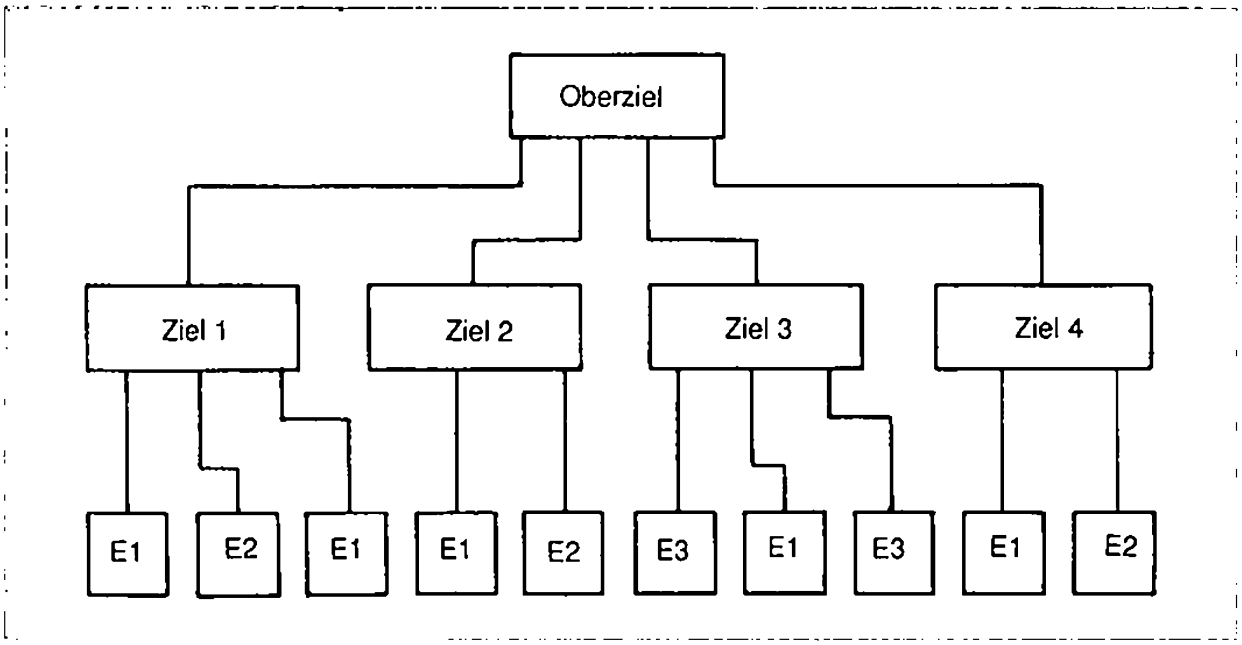

Abbildung 13-7: Strukturdiagramm mit Sprachelementen E1, E2, E3

Baustein Eingabekontrolle

Wie wäre es mit einem universellen Baustein zur Eingabekontrolle, nach allen Regeln der Kunst geschrieben? Die Prozeduren in den vorigen Kapiteln waren jeweils auf die Stelle im Programm bezogen, für die sie gebraucht wurden. Sie können sie jetzt parametrisieren und erhalten, fast ohne Veränderung, eine allgemein verwendbare Routine. Um *MenueEingabe3* aus Kapitel 11 in eine allgemeine Prozedur zur Überprüfung von Char-Eingaben umzuwandeln, müssen Sie zunächst dafür sorgen, daß im übergeordneten Programm eine Typvereinbarung existiert:

```
TYPE
  TypZeichenSet = SET OF Char;
```

Solche Deklarationen können eventuell mit in eine Baukastendatei aufgenommen werden. In Turbo-Pascal ist das unproblematisch, da es mehrere Typdeklarationsabschnitte geben darf. Alles andere bereitet keine Schwierigkeiten mehr:

```
PROCEDURE CharEingabe
  (VAR Zeichen          : Char;
   ZulaessigeEingaben : TypZeichenSet);

{Deklarationsteil wie auf S. 178}

BEGIN
  Zeichen := ReadKey;      {Oder: Read (kbd, Zeichen);}
  Write (Zeichen);
```

```
  WHILE NOT (Zeichen IN ZulaessigeEingaben)
                                {Hier geändert!}
  DO
    BEGIN
      Write (Fehlermeldung);
      VorFehlermeldungZurueckgehen;
      Zeichen := ReadKey; {Oder: Read (kbd, Zeichen);}
      Write (Zeichen);
      LoescheFehlermeldung;
    END;
                                {Hier geändert!}
END;
```

Die Zuweisung von *Zeichen* an *Anlass*, die ursprünglich als letzte Anweisung in der Prozedur stand, ist jetzt überflüssig. Das eingelesene Zeichen kommt im Endeffekt auf den Aktualparameter. Der Aufruf von *MenueEingabe* in Kapitel 11 kann dann zum Beispiel ausgetauscht werden gegen:

```
CharEingabe (Anlass, ['G','g','F','f','K','k','P','p','E','e']);
```

call by value und call by reference

Für einen Werteparameter dürfen, wie Sie hier sehen, Werte des betreffenden Typs in beliebiger Form stehen. Es wird daher auch von einem Aufruf „über den Wert" (call by value) gesprochen. Beim Variablenparameter ist die Nennung der Variablen, mit der die Prozedur arbeiten soll, zwangsläufig. Der Aufruf geschieht „über den Namen" oder „durch Bezugnahme auf eine Variable" (call by name beziehungsweise call by reference).

Vier weitere nützliche Bausteine

Der Baustein Zentrieren

Um Programme etwas ansprechender zu gestalten, ist eine Reihe weiterer Bausteine nützlich. Zum Beispiel rückt die Prozedur *Zentrieren* ein beliebiges *Textstueck* in die Mitte der Zeile:

```
PROCEDURE Zentrieren (Textstueck: TypNormalstring);
CONST
  Zeilenbreite = 80;
VAR
  AnzahlLeerzeichenInZeile,
  AnzahlLeerzeichenVorne, i : Integer;
```

```
BEGIN
  AnzahlLeerzeichenInZeile :=  Zeilenbreite - Length (Textstueck);
  AnzahlLeerzeichenVorne   :=  AnzahlLeerzeichenInZeile DIV 2;
  FOR i:= 1 TO AnzahlLeerzeichenVorne DO write (' ');
  Write (Textstueck);
END (* von Zentrieren *);
```

Da *Textstueck* ein Werteparameter ist, können beim Aufruf STRING-Werte eingesetzt werden (zum Beispiel *Zentrieren ('Guten Morgen!');*).

STRING-Typen

Bei diesem Beispiel stoßen Sie auf eine kleine Unannehmlichkeit: STRING [80] zum Beispiel gilt nicht als Typname, wie er bei der Parameterdefinition gefordert ist, sondern bereits als zusammengesetzte Typbeschreibung. STRING darf zwar in Turbo-Pascal ab Version 4.0 auch allein stehen (und bedeutet dann einen STRING der maximal zulässigen Länge 255), aber die vordefinierte Standardlänge ist nicht immer die richtige. Im allgemeinen sinnvoller (vorgeschrieben in Turbo-Pascal bis zur Version 3.x) ist, sich einen Typnamen für STRINGs der gewünschten Länge zu definieren. Vor der Deklaration von *Zentrieren* muß im Programm der Typ *Normalstring* vereinbart werden, etwa durch:

```
TYPE
  Normalstring = STRING [80];
```

Der Baustein SchreibeAb (Zeile, Spalte)

Vertauschen Sie bei GoToXY immer noch gelegentlich Zeile und Spalte? Dann können Sie sich mit der folgenden winzigen Prozedur behelfen, die die Originalanweisung an die allgemeinen Sprachgewohnheiten (erst die Zeile, dann die Spalte) anpaßt. Nebenbei gewährleistet sie, daß der Bildschirmbereich nicht überschritten wird:

```
PROCEDURE SchreibeAb (Zeile, Spalte : Integer);
CONST
  BildschirmHoehe  = 25;
  BildschirmBreite = 80;
BEGIN
  IF (Zeile <= BildschirmHoehe) AND (Spalte <= BildschirmBreite)
    THEN
      GoToXY (Spalte, Zeile);
END;
```

Der Aufruf *SchreibeAb (23, 4);* setzt dann den Cursor auf Zeile 23 und Spalte 4.

```
1 | Erwin Lindemann          | Pascalstr. 123, 4630 Bochum
2 | Michaela Meyer           | Modulastr. 4, 6500 Mainz
3 | Christian Programmsky    | Diskettenweg 43, 5090 Leverkusen
4 | Gabriele Schulze-Maier   | Blumenstr. 24, 6000 Frankfurt
```

Abbildung 13-8: Liste mit linksbündigen Texten

Der Baustein Linksbuendig

Es ist einfach, einen STRING rechtsbündig auf vorgegebener Breite zu schreiben, indem Sie die Breite hinter dem Formatierungs-Doppelpunkt nennen (zum Beispiel *Write (Textstueck: 40)*). Aber stellen Sie sich vor, Sie möchten eine Liste von Namen und Adressen auf den Bildschirm bringen: Dann müssen die Texte linksbündig in einem bestimmten Raum stehen.

Sie erreichen dies problemlos mit einer dafür geschaffenen Prozedur:

```
PROCEDURE Linksbuendig
  (Textstueck : Normalstring;
   Breite     : Integer);

BEGIN
  IF Length (Textstueck) > Breite
    THEN
      BEGIN
         Textstueck := Copy (Textstueck, 1, Breite);
         Write (Textstueck);
      END
    ELSE
      BEGIN
         Write (Textstueck);
         Write (''':Breite-Length (Textstueck));
      END;
END;
```

Wenn das *Textstueck* zu lang für seinen Platz ist, wird es mittels der vordefinierten Copy-Funktion gerade auf die zulässige *Breite* gestutzt und ausgegeben. Paßt es ohne Abschneiden, so wird der Text hingeschrieben und der Rest des vorgesehenen Platzes mit Leerzeichen aufgefüllt. Dabei wird der Trick angewendet, den leeren STRING '' auf die erforderliche Zahl von Zeichen zu formatieren:

```
Write (''':Breite-Length (Textstueck));
```

Der Baustein EroeffneZumLesen

Mittels der folgenden kleinen Prozedur können Sie nachschauen, ob auf der Diskette oder in einem Verzeichnis unter einem bestimmten Dateinamen eine Datei existiert. Dazu benötigen Sie einen weiteren Compilerbefehl. Normalerweise sorgt das Pascal-System dafür, daß ein Programm irregulär endet, falls ein Diskettenzugriff scheitert (zum Beispiel weil die Diskette fehlt oder der Dateiname ungültig war). Mit anderen Worten, das Programm stürzt ab, statt ohne gültige Daten weiterzulaufen. Diese übliche Kontrolle der Eingabe- und Ausgabevorgänge ist mit der Compileroption $I- („I" für Input/Output, I/O) aufhebbar. Allerdings muß man dann das Programm so gestalten, daß es selbst auf das Mißlingen von I/O-Operationen reagieren kann.

Zur Abfrage des Erfolges eines I/O-Vorgangs stellt Turbo-Pascal die Funktion *IOResult* zur Verfügung. In der Prozedur *EroeffneZumLesen* wird nach Abschaltung der I/O-Kontrolle versucht, eine Datei mit Reset zu eröffnen. IOResult trägt den Wert 0, wenn die Dateioperation gelingt. Scheitert die Eröffnung, steht eine Fehlernummer (eine ganze Zahl größer als 0) auf IOResult (welche Fehlernummer für welches Vorkommnis hängt von der Pascal-Version ab). Nach außen signalisiert dann der boolesche Variablenparameter nur, ob die Eröffnung erfolgreich verlief oder nicht. Damit im übrigen Programm wieder das Pascal-System für die Notbremse beim Auftreten von I/O-Fehlern sorgt, wird zum Schluß die automatische I/O-Kontrolle mit $I+ wieder eingeschaltet.

```
PROCEDURE  EroeffneZumLesen;
  (VAR Datei       : Text;
   Dateinamen      : Normalstring;
   VAR erfolgreich: Boolean);

BEGIN
  {$I- }
  Assign (Datei, Dateinamen) ;
  Reset (Datei);
  IF IOResult = 0
    THEN erfolgreich := true
    ELSE erfolgreich := false
  {$I+ }
END;
```

Mit der Prozedur läßt sich nicht nur feststellen, ob es eine bestimmte Datei auf einer Diskette gibt. Sie können die Prozedur auch in eine Schleife einbauen, die von den Benutzern des Programms die erneute Eingabe einer Dateibezeichnung verlangt, solange *erfolgreich* weiter *false* bleibt. Auf diese Weise erhalten Sie eine Art Eingabekontrolle für Dateinamen. Eventuell kann auch das Programm, wenn der Zugriff scheitert, selbst den Dateinamen verändern, zum Beispiel eine Laufwerksbezeichnung (A:, B:) vorne anfügen oder eine Namenserweiterung (.TXT, .PAS) hinten ergänzen, und dann erneut versuchen, die Datei zu eröffnen.

Parameter vom Typ Text

Wie am letzten Beispiel deutlich wird, können auch Parameter vom Standardtyp Text verwendet werden, allerdings nur Variablenparameter. Die Einschränkung ist erklärlich, denn sonst müßte eine Prozedur in der Lage sein, sich eine interne Kopie einer ganzen Datei anzufertigen.

UNITs

Das oben geschilderte Verfahren, immer wieder verwendete Prozeduren in eine Datei BAUKAST.INC zu stecken, die in beliebige Programme hineinkopiert oder per Include-Anweisung vom Compiler aufgenommen werden kann, hat den Nachteil, daß die Prozeduren immer wieder neu kompiliert werden. Dabei steht längst fest, daß sie keine Syntaxfehler mehr enthalten. Bei längeren Programmen passiert es auf diese Weise leicht, daß hunderte von Kilobyte Programmtext ständig neu kompiliert werden, obwohl nur einige wenige Programmzeilen hinzugefügt oder geändert werden. Abhilfe schafft hier die Einführung von UNITs, Moduln, wie sie in Turbo-Pascal ab Version 4.0 und in Quick-Pascal, aber auch im alten UCSD-Pascal zur Verfügung stehen.

Import und Export

Im formalen Aufbau ähnelt eine UNIT einem selbständigen Programm mit Kopf, Deklarationsteil mit Konstanten, Typen, Variablen und Prozeduren sowie einem Anweisungsteil und wird wie ein solches kompiliert. Eine UNIT verrichtet jedoch nicht als ganzes eine eigene Aufgabe, sondern stellt in ihm deklarierte Konstante, Typen, Variable und Prozeduren für Programme und andere UNITs bereit. In Anlehnung an die beim Pascal-Abkömmling Modula-2 gebräuchliche Redeweise wird da-

von gesprochen, daß eine UNIT bestimmte Bezeichner (Namen von Konstanten, Typen und so fort) exportiert und ein Programm beziehungsweise eine andere UNIT diese dann importieren kann. Neben dem rein internen IMPLEMENTATION-Abschnitt gehört daher zu jeder UNIT ein INTERFACE-Teil, in dem die potentiellen Export-Objekte aufgezählt werden. Jedes Programm (oder auch eine andere UNIT), das mittels USE eine UNIT angefordert hat, kann die Objekte von dessen INTERFACE-Teil aufrufen und gebrauchen, als wären sie seine eigenen.

Die UNIT Baukast

Wenn Sie über Turbo-Pascal ab Version 4.0 oder über Quick-Pascal verfügen, ist es zum Beispiel sinnvoll, daß Sie sich anstelle der Include-Datei BAUKAST.INC ein UNIT *Baukast* (die den Dateinamen BAUKAST.PAS tragen muß) einrichten. Bei der Anforderung dieser UNITs müssen Sie beachten, daß jedes Programm, welches *Baukast* verwendet, auch die in *Baukast* benutzte UNIT Crt anfordern muß. Wie erhalten Sie nun dies gewünschte UNIT? Zunächst schreiben Sie den UNIT-Kopf und den folgenden INTERFACE-Teil. Die hier genannten Prozeduren, Typen und so fort können anderen Programmen und UNITs zur Verfügung gestellt werden:

```
UNIT Baukast;
INTERFACE
  USES Crt, Printer;
  TYPE
    TYPZeichenset = SET OF Char;
    Normalstring  = String [80];
  PROCEDURE CharEingabe
       (VAR Zeichen          : Char;
        ZulaessigeEingaben : TypZeichenset);
  PROCEDURE SchreibeAb (Zeile, Spalte : Integer);
  PROCEDURE Linksbuendig
       (Textstueck           : Normalstring;
        Breite               : Integer);
  PROCEDURE EroeffneZumLesen;
       (VAR Datei            : Text;
        Dateinamen           : Normalstring;
        VAR erfolgreich      : Boolean);
```

Der INTERFACE-Teil endet mit dem Schlüsselwort IMPLEMENTATION (ohne Semikolon). Danach wird die Innenwelt der UNIT beschrieben: Bei Bedarf können weitere Konstanten, Typen, Variable und Prozeduren verein-

bart werden. Außerdem wird hier die vollständige Deklaration aller Prozeduren nachgeliefert. Die Prozedurköpfe werden dabei ein zweites Mal aufgeführt. Schließlich folgt noch der (oft leere) Anweisungsteil der UNIT, der durch BEGIN END umrahmt und mit einem Punkt abgeschlossen ist. Seine Anweisungen dienen ausschließlich der Initialisierung von Variablen. Sie werden lediglich einmal ausgeführt, und zwar unmittelbar beim Aufruf des Programms, welches die UNIT mit einer USES-Klausel anfordert. Zum Beispiel können Sie, je nach Wunsch im INTERFACE- oder im IMPLEMENTATION-Teil, eine Variable *Kleinbuchstaben* mit *TypZeichenset* deklarieren, die Sie mit folgender Anweisung im Anweisungsteil des UNIT ihrem Sinn entsprechend initialisieren:

```
Kleinbuchstaben : = ['a'..'z'  'ä', 'ö', 'ü', 'ß'];
```

Die Bedeutung von UNITs

UNITs bilden das geeignete Mittel, um vorgefertigte Programmelemente (Tools) auf Vorrat zu halten. Da die von ihnen zur Verfügung gestellten Prozeduren verwendet werden können, auch wenn sie nur in schon kompilierter Form vorliegen, können sie kommerziell vertrieben werden, ohne daß ihre interne Funktion offengelegt werden muß. Turbo- und Quick-Pascal machen selbst reichlich Gebrauch von dieser Möglichkeit. Neben den UNITs Crt und Printer, die in allen Programmen dieses Buches angefordert werden, gehören unter anderem UNITs zur Grafikverwendung und zur Nutzung von DOS-Systemaufrufen zum Lieferumfang von Turbo- und Quick-Pascal. Der Hauptnachteil der UNITs besteht darin, daß sie in Standard-Pascal noch nicht vorgesehen wären und es daher Probleme bei der Übertragung auf andere Systeme geben kann.

Funktionen

Funktionen sind eine spezielle Sorte von Unterprogrammen. Für Funktionen ist charakteristisch, daß sie ein Ergebnis hervorbringen, das sich unmittelbar in einem Pascal-Programm wie ein Wert verwenden läßt, zum Beispiel auf der rechten Seite einer Wertzuweisung. Mit einer Reihe von vordefinierten Funktionen wurde bereits gearbeitet:

- Round (Dezimalzahl) ergibt die zum Real-Wert *Dezimalzahl* gehörende gerundete Integer-Zahl (zum Beispiel ergibt Round (17.9) die Zahl 18).
- Trunc (Dezimalzahl) schneidet die Nachkommastellen eines Real-Wertes ab (trunc (17.9) ergibt die Zahl 17).
- Ord (Wert) bezeichnet die Ordnungsnummer entsprechend der Werteliste eines Aufzählungstyps (zum Beispiel ist ord ('a') gleich 97).
- Chr (Nummer) ist eine auf Char-Werte beschränkte Umkehrung von ord. Es kommt das zu *Nummer* gehörige Zeichen gemäß der Char-Liste heraus (Zum Beispiel bedeutet chr (97) das Zeichen 'a').

Diese vier Funktionen sind die vier Transferfunktionen von Standard-Pascal. Sie heißen so, weil sie Werte von einem Typ zu einem anderem übertragen. Im letzten Kapitel wurden auch die Nachfolger- und Vorgängerfunktion für Aufzählungstypen vorgestellt (succ und pred). Auch der boolesche Ausdruck *eof (Datei)*, durch den das Programm das Ende einer Datei feststellt, stellt einen Aufruf der Funktion eof dar. Ähnlich läßt sich mit *EoLn (Datei)* prüfen, ob in einer Textdatei gerade ein Zeilenende gelesen wurde.

KeyPressed

In Turbo-Pascal gibt es die vordefinierte, parameterlose boolesche Funktion *KeyPressed*. Mit ihrer Hilfe läßt sich feststellen, ob irgendeine Taste gedrückt wurde, um nur in diesem Fall ein Zeichen von der Tastatur einzulesen. Auf diese Weise kann zum Beispiel eine Schleife immer weiter laufen, solange keine Taste betätigt wird:

```
WHILE NOT Keypressed DO    { ... };
{Weiterdrucken}
```

Wenn das Zeichen unter Verwendung einer Leseanweisung vom Tastaturpuffer geholt wird, zeigt KeyPressed wieder false.

Mathematische Funktionen in Pascal

Den Absolutbetrag einer Zahl, in der Mathematik | x | geschrieben, erhält man in Pascal mit:

```
abs (x)
```

Testen Sie die Funktion mit einem kleinen Programm, das zu jeder eingegebenen (auch negativen) Zahl ihren Betrag anzeigt. Als Abkürzung für x*x dürfen Sie, um eine Zahl zu quadrieren, auch schreiben:

```
sqr (x)
```

(gelesen wie englisch „square", Quadrat). Die Umkehrung des Quadrierens ist das Ziehen der Quadratwurzel, symbolisiert durch das Wurzelzeichen √. √a zu bestimmen, heißt in der Mathematik, die Zahl zu suchen, die mit sich selbst multipliziert gerade a ergibt. Zum Beispiel ist √9 gleich 3 (denn 3*3=9) oder √2 ungefähr 1.414... Das Pascal-System führt diese Operation aus, wenn die Funktion *sqrt* (square root, „Quadratwurzel") aufgerufen wird:

```
sqrt (a)
```

Es gibt in der Mathematik eine Sorte von Funktionen, die transzendent heißen. Die wichtigsten gehören zum Standard von Pascal: *sin* (Sinus), *cos* (Cosinus), *ln* (der natürliche Logarithmus zur Basis e), *arctan* (Arcustangens), *exp* (Exponens). In Turbo-Pascal existieren noch zahlreiche weitere vordefinierte Funktionen, die zum Teil schon erwähnt wurden oder die Sie im Handbuch nachschlagen können. Aber es ist auch nicht schwer, eigene Funktionen zu deklarieren.

Selbstdefinierte Funktionen

Deklaration

Abgesehen davon, daß sie mit dem Schlüsselwort FUNCTION anstelle von PROCEDURE eingeleitet wird, unterscheidet sich die Syntax der Funktionsdeklaration nur wenig von einer Prozedurdeklaration. Entscheidend ist, daß die Funktion nachher quasi auf ihrem Namen einen Wert – das Funktionsergebnis – trägt. Daher wird der Kopf der Deklaration abgeschlossen mit der Angabe des Ergebnistyps für die Funktion. In Standard-Pascal sind als Ergebnistyp nur einfache Typen (Real, Integer, Char, Boolean sowie Teilbereiche), in Turbo-Pascal außerdem STRING zugelassen. Im Anweisungsteil der Funktion muß irgendwann ein Wert – das Funktionsergebnis – dem Funktionsnamen zugewiesen werden. Insofern wird der Funktionsname wie ein Variablenname gebraucht. Jedoch darf er nicht auf der rechten Seite einer Wertzuweisung oder als Teil eines Ausdrucks erscheinen. Eine Funktion darf wie eine Prozedur einen (oder keinen oder mehrere) Werte- oder Variablenparameter besitzen. Da sie aber auf ihrem eigenen Namen Ergebnisse nach außen tragen kann, sollten in der Regel keine Variablenparameter zum selben Zweck verwendet werden.

Betrachten Sie das folgende Beispiel einer booleschen Funktion, das heißt einer Funktion mit dem Ergebnistyp Boolean. Gerade solche Funktionen können sich, ähnlich wie zum Beispiel die vordefinierte eof-Funktion, beim Programmieren als sehr praktisch erweisen. Die Funktion *UrlaubsgeldReicht* würde sich zum Beispiel eignen, um Entscheidungen zu beeinflussen oder Signale für das Verhalten im Urlaub zu setzen:

Eine boolesche Funktion

```
FUNCTION UrlaubsgeldReicht
  (MomentanerKassenstand, MitgenommenesGeld : Real;
   GesamtUrlaubslaenge, RestUrlaubslaenge    : Integer)
   : Boolean;

VAR
  AusgegebenesGeld,
  BisherigerTagesverbrauch    : Real;
  VerstricheneUrlaubstage     : Integer;

BEGIN
  VerstricheneUrlaubstage := GesamtUrlaubslaenge -
                             RestUrlaubslaenge;
  AusgegebenesGeld := MitgenommenesGeld - MomentanerKassenstand;
  BisherigerTagesverbrauch := AusgegebenesGeld /
                             VerstricheneUrlaubstage;
  IF Resturlaubslaenge * BisherigerTagesverbrauch <
     MomentanerKassenstand
    THEN UrlaubsgeldReicht := true
    ELSE UrlaubsgeldReicht := false;
END;
```

Eine Anwendung einfachster Form wäre die Anweisung:

```
IF NOT UrlaubsgeldReicht (300.78, 800, 14, 6)
  THEN WriteLn ('Sparsamer leben oder Geld beschaffen! ');
```

Ein kleines Beispiel aus der Statistik ist die Funktion *Mittel*: zwei Zahlen werden gemittelt, indem sie addiert werden und das Ergebnis durch 2 dividiert wird. Sowohl die beiden Parameter als auch das Ergebnis sind dabei vom Typ Real.

Mittelwertsfunktion

```
FUNCTION Mittel (Zahl1, Zahl2:Real) : Real;
VAR
  Summe : Real;
BEGIN
  Summe := Zahl1 + Zahl2;
  Mittel := Summe/2;
END;
```

Zur geschachtelten Verwendung von Prozeduren

Eine Prozedur (oder eine Funktion) kann, wie Sie wissen, genauso wie ein selbständiges Programm eigene Unterprozeduren und Unterfunktionen haben und diese verwenden. Sie hat darüber hinaus Zugriff auf vorher deklarierte Prozeduren:

Merkzettel

> Eine Prozedur bzw. eine Funktion kann alle Prozeduren und Funktionen aufrufen, die entweder auf derselben oder einer höheren Ebene als sie stehen.

Im Programm *Kartenspiel* aus dem letzten Kapitel darf deshalb die Prozedur *BlattZeigen* die Prozedur *Anzeigen* rufen. Eine Unterprozedur von *BlattZeigen* könnte dasselbe tun. Allerdings muß in jedem Fall der Name der gerufenen Prozedur an dem Punkt des Programmtextes, an dem der Ruf ergeht, schon bekannt sein (sonst kommt die Fehlermeldung „nicht deklariert"). Der technische Grund dafür ist, daß Pascal-Compiler in der Lage sein sollen, in einem einzigen Durchgang das Programm in Maschinensprache zu übersetzen. Deshalb „überblicken" Sie nicht, ob eine Prozedur später noch deklariert wird.

FORWARD-Deklaration von Prozeduren und Funktionen

Es gibt jedoch Fälle, in denen sich Prozeduren oder Funktionen gegenseitig beispringen müssen. Dann muß die zuerst im Programmtext auftauchende Prozedur auch die später deklarierte verwenden können. Als Ausweg für diese Situation ist in Pascal vorgesehen, von der zweiten Prozedur (oder von beiden) vorneweg erst einmal den Prozedurkopf zu nennen und damit anzukündigen: „Deklaration folgt noch – Fehlermeldung nicht erforderlich". Dazu wird zunächst der vollständige Prozedurkopf (samt eventuellen Parametern) mit der Ergänzung FORWARD hingeschrieben und so nach „vorwärts" verwiesen. Irgendwann im weiteren Verlauf des Textes wird der Kopf der Prozedur ohne Parameterangaben wiederholt und der Deklarations- und Anweisungsteil hingeschrieben. Zum Beispiel sähe die FORWARD-Deklaration bei *BlattZeigen* in der letzten Fassung so aus:

```
PROCEDURE BlattZeigen (Kartengruppe: TypKartenSet); FORWARD;
```

Die eigentliche Definition käme dann irgendwann später im Programm und würde lediglich eingeleitet durch

PROCEDURE BlattZeigen; (ohne Wiederholung der Parameter).

Eine Prozedur oder eine Funktion darf sich auch selbst aufrufen (beziehungsweise eine zweite Prozedur, die dann wieder auf die erste zurückgreift). In diesem Fall rekurriert sie auf sich selbst, das heißt, sie läuft auf sich selbst zurück. Es wird von einer rekursiven Verwendung der Prozedur oder von einer Rekursion gesprochen.

Rekursion

Gewiß haben Sie es schon einmal ausprobiert: Wenn zwei Spiegel fast parallel gestellt werden, spiegelt sich der erste im zweiten, der zweite sendet sein Konterfei samt Spiegelbild des ersten zu diesem zurück, der setzt das Spiel fort. In einem Augenblick pflanzt sich die Reihe der Bilder von gespiegelten Spiegelbildern fort, bis sie scheinbar im Unendlichen verschwimmt. Das ist ein ganz alltägliches Beispiel für das, was geschieht, wenn sich zwei Funktionen, hier zwei Spiegelungen, wechselseitig ansprechen. Die faszinierenden Phänomene, die herauskommen, stellen die Fähigkeit, ihr Zustandekommen zu begreifen, manchmal auf eine harte Probe. Die Rekursion bei Prozeduren macht es wie die Spiegel möglich, mit geringen Mitteln die erstaunlichsten Effekte hervorzaubern. Eine Reihe fortgeschrittener Programmiertechniken lassen sich ohne rekursive Prozeduren kaum bewerkstelligen. Freilich läßt sich auch mit keinem anderen Mittel so leicht ein Systemabsturz heraufbeschwören. Es ist eine der interessanten und wichtigsten, wenn auch nicht ungefährlichen Eigenschaften von Pascal, daß im Gegensatz zu früheren Programmiersprachen Rekursion grundsätzlich zugelassen wird.

Rekursion und Wiederholungsstruktur

Vom Sprachgefühl her scheint eine Rekursion zunächst durchaus einfach zu sein. Zum Beispiel könnte eine einfache Prozedur zur Eingabekontrolle folgende Form haben:

```
PROCEDURE Eingeben
  (VAR Zeichen          : Char;
   ZulaessigerBereich : TypZeichenSet);
BEGIN
   Zeichen := ReadKey
     IF NOT (Zeichen IN ZulaessigerBereich)
       THEN Eingeben (Zeichen, ZulaessigerBereich);
END;
```

Die Prozedur würde ein Zeichen einlesen und nachsehen, ob es im SET *ZulaessigerBereich* enthalten ist. Falls ja, gäbe sie das Zeichen auf dem Variablenparameter nach außen. Wenn nein – nun, dann müßte man es wieder eingeben. Die Prozedur riefe sich selbst erneut auf. Vielleicht kommt Ihnen diese rekursive Methode, etwas ähnliches erneut geschehen zu lassen, sogar einfacher vor, als eine Wiederholungsstruktur anzuwenden. Dennoch handelt es sich um ein Beispiel, wie Rekursion *nicht* angewandt werden soll! Denn bei jedem Aufruf einer rekursiven Prozedur (wie auch sonst bei jedem Prozeduraufruf) wird eine Kopie des internen Variablensatzes der Prozedur im Speicher abgelegt. Nach dem Verlassen des jeweiligen Prozedurlaufs gibt das Pascal-System den betreffenden Speicherplatz wieder frei. Wenn beim Abarbeiten einer rekursiven Prozedur immer weiter (rekursive) Aufrufe hinzukommen, stapeln sich die benötigten Variablen immer höher. Ohne Vorsichtsmaßnahmen ergibt sich schnell eine Reihe in die Unendlichkeit wie bei den Spiegeln, aber: Irgendwann ist die Grenze des Speichers erreicht und das Programm stürzt ab.

Grundsätzlich soll die Rekursion nur verwendet werden, wenn man tatsächlich an der Erzeugung lauter verschiedener, tief „gestapelter", dynamischer (das heißt erst zur Laufzeit entstehender) Variablen interessiert ist. Immer, wenn offensichtlich mit einer „flachen" Wiederholungsstruktur und einer im wesentlichen festen (statischen) Variablenausstattung ein Problem gelöst werden kann, sollte dieser Weg beschritten werden.

Intern muß und kann das Pascal-System jeden rekursiven Programmablauf in irgendeine Form einer Wiederholungsstruktur übersetzen. Wie auch mit großen statischen Objekten umgegangen werden kann, ohne zu rekursiven Verfahren Zuflucht zu nehmen, behandelt das

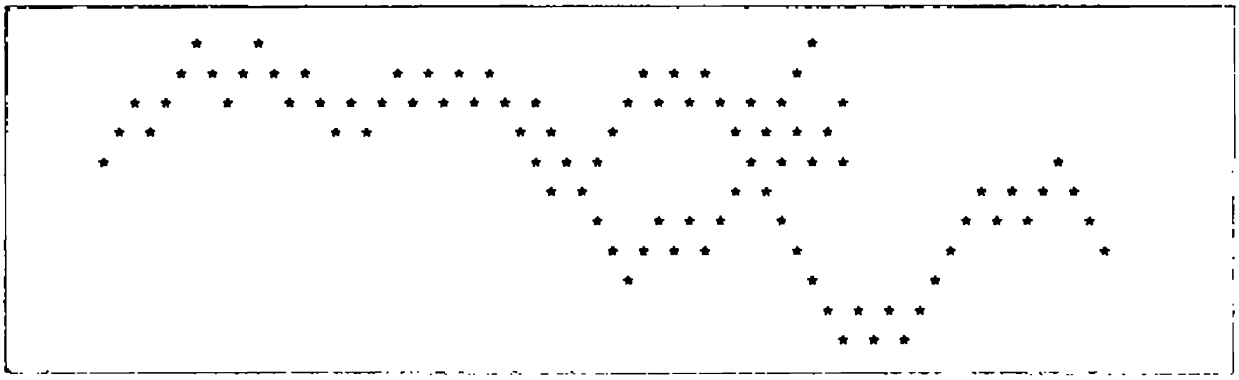

Abbildung 13-9: „Irrfahrt"

folgende Kapitel. Die nähere Betrachtung dynamischer Datenstrukturen wie auch der Raffinessen rekursiver und verschachtelter Prozeduraufrufe übersteigt leider den Rahmen dieses Buchs.

Aufgaben

1. Zeichnen Sie das Syntaxdiagramm für die Deklaration von Funktionen.

2. Führen Sie die im Text angedeutete Abwandlung von *KartenSpiel* aus. Zwei Personen sollen abwechselnd je eine Karte ziehen, wer jeweils die höhere hat, darf beide behalten. Gewonnen hat, wer am Schluß die höchste Punktzahl auf der Hand hält.

3. Schreiben Sie eine universell verwendbare, modulartige Prozedur, an die ein STRING-Parameter *Wort* und eine Anfangsposition für den Bildschirm übergeben wird. Die Prozedur soll dann das *Wort* – sofern es aus zulässigen Zeichen besteht und der Platz reicht – in groß gerasterten Figuren auf den Schirm schreiben (vergleiche Abb. 11-2).

4. Schreiben Sie sich einen Prozedur-Baustein, der vom Anwender die erneute Eingabe eines Dateinamens (oder eine Abbruchbestätigung) so oft verlangt, wie die Dateieröffnung noch nicht gelungen ist. Testen Sie seine Funktionstüchtigkeit aus.

5. Das folgende Programm mit der rekursiven Prozedur *Zufallsschritt* läßt ein Sternchen merkwürdige Wege auf dem Bildschirm vollführen (Abb. 13-9). Auf Knopfdruck (mit der Leertaste) wandert es sogar wieder exakt seinen Weg zurück und tanzt neu los (daran erkennen Sie, daß alle alten Orte gespeichert sind).

```
PROGRAM GrosseIrrfahrt;
USES crt, printer;
CONST
  Bildhoehe     = 24;
  Verzoegerung = 100;
PROCEDURE ProgrammErklaeren;
  BEGIN
    ClrScr;
    WriteLn ('Dieses Programm läßt ein Sternchen wandern.');
    WriteLn ('Mit der LEERTASTE können Sie zurückwandern.');
    WriteLn ('Abbrechen mit CTRL (Strg) und BREAK (Pause)!');
  END;
```

Prozeduren

```
PROCEDURE Zufallsschritt (Spalte, Zeile: Integer);
VAR
  Zufallszahl      : Integer;
  Zeichen          : Char;
BEGIN
  delay (Verzoegerung);
  Zeichen := 'x';
  IF KeyPressed THEN Zeichen := Readkey)
                {Oder: Read (kbd, Zeichen);}
  IF (Zeile < Bildhoehe) AND (Zeile > 5)  AND (Spalte < 80) AND
     (Zeichen <> ' ')
                {Abbruch eines Weges, falls die Bildgrenzen
                 erreicht sind oder Leertaste betätigt wird.}
    THEN
      BEGIN
        GoToXY (Spalte, Zeile);
        Write ('*');
        delay (Verzoegerung);
        Zufallszahl := Random (2);
        IF Zufallszahl = 1
          THEN
            BEGIN
              { zuerst nach unten gehen: }
              Zufallsschritt (Spalte + 1, Zeile + 1);
              { zurückgehen auf alten Punkt: }
              GoToXY (Spalte, Zeile);
              delay (Verzoegerung);
              { von hier jetzt nach oben gehen }
              Zufallsschritt (Spalte + 1, Zeile - 1);
            END
          ELSE
            BEGIN
              { zuerst nach oben gehen }
              Zufallsschritt (Spalte + 1, Zeile - 1);
              { zurückgehen auf alten Punkt: }
              GoToXY (Spalte, Zeile);
              delay (Verzoegerung);
              { von hier jetzt nach oben gehen }
              Zufallsschritt (Spalte + 1, Zeile + 1);
            END;
      END;
END;
BEGIN
  ProgrammErklaeren;
  Randomize;
  Zufallsschritt (1, 15);
END.
```

14 RECORD, ARRAY und FILE

Die Abbildung realer Objekte im Datenverbund – Die WITH-Anweisung – Speichern auf Textdateien – ARRAYs – Geschachtelte Datentypen – Variante Records – Die Datenstruktur FILE OF für den Direktzugriff – Sortieren – Weitere Beispiele – Aufgaben

Wenn ein Programm sich mit „großen" Objekten befaßt, müssen die Datenstrukturen die Komplexität widerspiegeln. Dieses Kapitel stellt die komplexen Datenstrukturen RECORD, ARRAY und FILE vor.

Eine Sichtweise ist folgende: Das grundlegende Mittel, um in einem Pascal-Programm Bedeutungen auszudrücken, besteht in der Vergabe von „sprechenden" Namen. Dies gilt für Variablen ebenso wie für Programme und Prozeduren. Und so wie eine Prozedurvereinbarung aus einer Kette einzelner Anweisungen ein strukturiertes Ganzes macht, so kann und soll nun auch der strukturierte Zusammenhang von Einzeldaten in Pascal durch die Zusammenfassung zu größeren Einheiten ausdrücklich benannt werden.

Die Abbildung realer Objekte im Datenverbund

Zum Beispiel werden in einem Betrieb für die Lohnbuchhaltung die Daten der Angestellten verarbeitet. Zu Zwekken der Gehaltsabrechnung sind folgende Einzeldaten vorhanden: Name, Vorname, Personalnummer, Gehaltsgruppe, Geschlecht, Alter, Kinderzahl, Familienstand, monatliche Arbeitszeit, Krankheitstage, Fehlzeiten, Anschrift, Kontonummer, Lohnpfändungsvermerke ...

Ein Datensatz-Typ

Wenn ein Programm eine Gehaltsabrechnung für eine Person durchführt, benötigt es die Gesamtheit der entsprechenden Daten. Diese Gesamtheit repräsentiert gleichsam die jeweilige Person im Hinblick auf den gegebenen Zweck und bildet den eigentlichen Gegenstand des Programms. Damit die Objektstruktur im Pascal-Programm auch zum Ausdruck kommt, deklarieren Sie nicht

lauter isolierte Variablen *Name, Vorname* und so weiter: mittels eines RECORD-Typs fügen Sie die verschiedenen einzelnen Angaben zu einem Datenverbund oder Datensatz (englisch Record) zusammen:

```
TYPE
  TypPersonDatensatz =
    RECORD
      Name                : STRING [30];
      Vorname             : STRING [30];
      Personalnummer      : Integer;
      Gehaltsgruppe       : STRING [5];
      Geschlecht          : (weiblich, maennlich);
      Alter               : Integer;
      Kinderzahl          : Integer;
      WochenArbeitszeit   : Real;
      Kontonummer         : STRING [20];
      Steuerklasse        : Char;
      Lohnpfaendung       : Boolean;
      Anschrift           : STRING[40];
    END;
```

RECORD-Deklaration

Sie sehen, wie ein RECORD-Typ deklariert wird: Nach dem Typnamen und dem Gleichheitszeichen steht das Schlüsselwort RECORD, auf das eine Aufzählung der Komponenten, der Bestandteile des Datensatzes, folgt. Das ganze wird durch END (ähnlich wie die Falliste bei CASE) abgeschlossen. Jede Komponente ist durch Angabe ihrer Bezeichnung und ihres Typs festgelegt. Die Typangabe darf sowohl ein Typname als auch eine Typbeschreibung (eine implizite Typvereinbarung) sein. Um eine implizite Typvereinbarung handelt es sich im Beispiel sowohl bei den STRING-Typen wie auch bei *(weiblich, maennlich)*. Denn die Geschlechtsbezeichnung ist hier ein selbstdefinierter Aufzählungstyp. Die implizite Form wurde nur der Kürze halber gewählt. Meist ist es vorteilhafter oder sogar unumgänglich, sich explizite Typen zu verschaffen, wie zum Beispiel:

```
TypGeschlecht = (weiblich, maennlich);
```

Bei STRING-Typen kann es sinnvoll sein, zuerst die Maximallängen als Konstanten zu vereinbaren:

```
CONST
  Namenslaenge        = 30;
  Gehaltsgruppenlaenge =  5;
```

```
Kontonummerlaenge     = 20;
Anschriftenlaenge     = 40;
```

In der Typbeschreibung werden dann die Konstantennamen für die Werte eingesetzt. Werden Prozedurparameter benötigt, müssen Sie eventuell sogar für jede der verschiedenen STRING-Längen einen gesonderten Typnamen definieren:

```
TYPE
  TypNamenString            =  STRING [Namenslaenge];
  TypGehaltsgruppenString =  STRING [Gehaltsgruppenlaenge];
                            { ... }
```

Schon an solchen allerersten Details zeigt sich, wie aufwendig ein Programm werden kann, das sich mit komplexen Objekte befaßt. Die folgenden Beispiele haben daher noch mehr als in den übrigen Kapiteln einen erläuternden und anregenden Charakter, als daß sie zur kompletten Ausführung empfohlen würden.

Komponentennamen

Für die Komponentennamen kann man beliebige Pascal-Namen verwenden. Ausnahmsweise schadet es diesmal nicht, wenn diese Namen an anderer Stelle noch eine andere Bedeutung haben. Die Komponente ist nichts Selbständiges, sondern erhält ihre Bedeutung aus dem Zusammenhang des Datensatzes.

Die Deklaration einer Record-Variable sieht wie eine gewöhnliche Variablendeklarationen aus. Es ist ihr nicht anzumerken, daß hier jede Variable für einen ganzen Datensatz steht (und entsprechend umfangreicher Speicherplatz reserviert wird):

```
VAR
  Person, Person1, Person2 : TypPersondatensatz;
```

Grafisches Schema

Datensätze werden oft schematisch durch eine Rechteckzeile dargestellt, in der einzelne Felder den Komponenten entsprechen (Abb. 14-1):

Für den Datensatz *Person* ist in das erste Feld der *Name* einzutragen, ins zweite Feld der Vorname und so fort.

Begriffe

Entsprechend der Vorstellung einer Tabelle mit mehreren Spalten bzw. Rubriken heißen die Komponenten eines Datensatzes (die Inhalte der Variablen) auch Datenfelder oder Datenelemente. Die Überschriften der Rubriken, das heißt die Komponenten des Datensatz-Typs, werden auch Attribute oder Datenfeldnamen genannt.

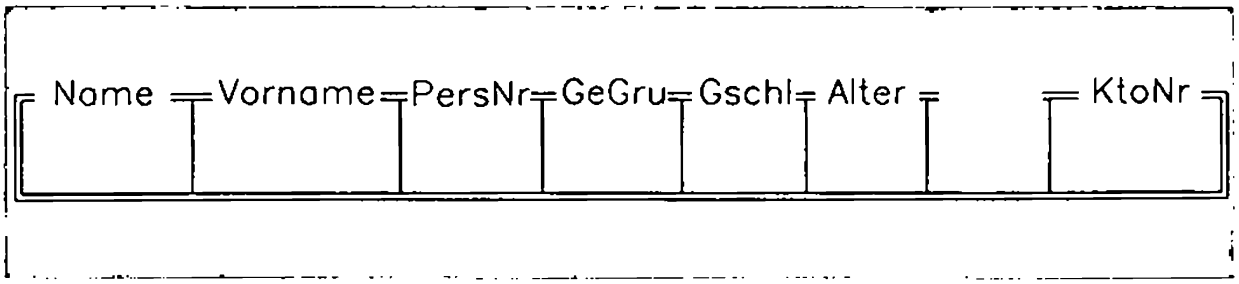

Abbildung 14-1: Schemaskizze des Datensatzaufbaus

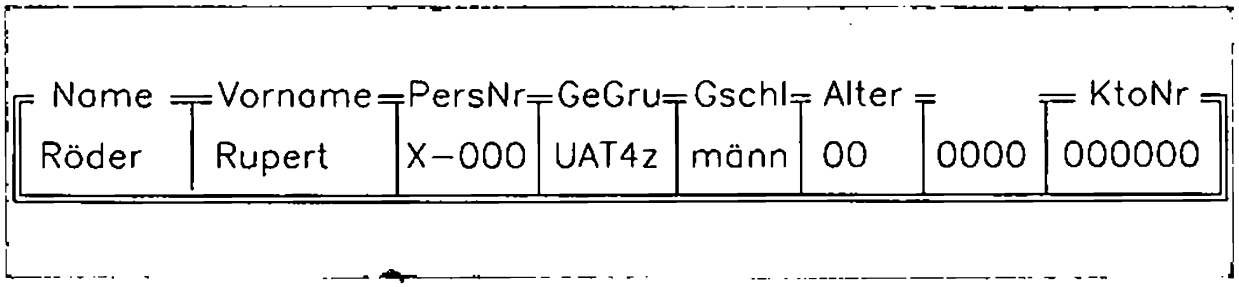

Abbildung 14-2: Skizze eines konkreten Datensatzes

Ansprechen der RECORD-Komponenten

Mit den Komponenten eines RECORDs kann wie mit Einzelvariablen gearbeitet werden. Um eine bestimmte Komponente anzusprechen, schreibt man den Namen der RECORD-Variablen hin, gefolgt von einem Punkt und dem Komponentennamen. Zum Beispiel bedeutet

```
Person2.Gehaltsgruppe
```

die *Gehaltsgruppe* im Datensatz *Person2*. Die nähere Bestimmung, von welcher *Gehaltsgruppe* die Rede ist (nämlich von der zu *Person2* gehörigen) wird auch Qualifizierung genannt. Hierzu ein paar durch Kommentarzeilen abgetrennte Beispiele:

```
WriteLn (Person.Name, ' ist ', Person.Alter, ' Jahre alt' );
        {--------------------}
IF Person.Lohnpfaendung THEN WriteLn ('Rücksprache nehmen');
        {--------------------}
IF Person1.Alter > Person2.Alter
  THEN
    WriteLn (Person1.Name, ' ist ',
             Person1.Alter - Person2.Alter,' Jahre älter als ',
             Person2.Name)
  ELSE
    IF  Person1.Alter  =  Person2.Alter
      THEN
        WriteLn (Person1.Name, ' ist ebenso alt wie ',
```

```
                Person2.Name)
    ELSE
      WriteLn (Person1.Name, ' ist ',
                 Person2.Alter - Person1.Alter,' Jahre jünger ',
                 als Person2.Name);
      {--------------------}
Person2.Gehaltsgruppe := 'IVa';
```

Da der Typ explizit angegeben wurde, können die Datensätze auch mit parametrischen Prozeduren bearbeitet werden. Zum Beispiel sei deklariert:

```
PROCEDURE FreudigesEreignis (VAR Person: TypPersonDatensatz);
BEGIN
   Person.Kinderzahl := Person.Kinderzahl + 1;
END;
```

Bei einer Geburt im Haus *Person1.Adresse* kann dann aufgerufen werden:

```
FreudigesEreignis (Person1);
```

Zwillinge von *Person2* erfordern einen Doppelaufruf:

```
FreudigesEreignis (Person2); FreudigesEreignis (Person2);
```

Eine Eingabeanweisung bildet den Kern der folgenden Prozedur:

```
PROCEDURE Befoerderung (VAR Person: TypPersonDatensatz);
BEGIN
  Write  (' Neue Gehaltsstufe von ');
  Write  (Person.Name, ', ', Person.Vorname);
  WriteLn;
  Write  ('Personalnummer:        ', Person.Personalnummer);
  WriteLn;
  Write  ('Bisherige Einstufung: ', Person.Gehaltsgruppe);
  Write  ('  neue Stufe: ');
  ReadLn (Person.Gehaltsgruppe);
END;
```

Die WITH-Anweisung

Wertzuweisung zwischen RECORD-Variablen

Wie bei allen anderen Variablen ist zwischen RECORDs, sofern sie vom selben Typ sind, eine Wertzuweisung möglich: Der Befehl *Person := Person1* überträgt zum Beispiel in einem Zug sämtliche Datenfelder, die zu *Person1* gehören, auf *Person*. Dies ist klarer als die getrennte Wert-

zuweisung für jedes Datenfeld und erspart darüber hinaus etliche Schreibarbeit.

WITH-Klausel Mit der sogenannten WITH-Anweisung läßt sich der Schreibaufwand weiter vermindern: Bei der von WITH abhängigen Anweisung werden alle Komponentennamen automatisch mit der angesprochenen RECORD-Variable verbunden. Auf diese Weise können Sie die oben angeführte Entscheidungsstruktur verkürzen:

```
WITH Person1 DO
  IF Alter > Person2.Alter
    THEN
      WriteLn (Name, ' ist ', Alter -  Person2.Alter,
              'Jahre älter als ', Person2.Name)
    ELSE
      IF Alter = Person2.Alter
        THEN WriteLn (Name, ' ist ebenso alt wie ', Person2.Name)
        ELSE
          WriteLn (Name, ' ist ', Person2.Alter - Alter,
                   ' Jahre jünger als ', Person2.Name);
```

Innerhalb der Reichweite der WITH-Klausel werden alle Komponentennamen, die unqualifiziert allein stehen, auf *Person1* bezogen. Bei diesem Verfahren muß genau darauf geachtet werden, ob es die Komponentennamen sonst noch als Variablennamen gibt – *Alter* oder *Name* dürfen hier nicht zugleich als selbständige Variablen vorkommen.

Wie üblich kann man durch einen BEGIN-END-Rahmen auch mehrere Anweisungen dem DO zuordnen und dadurch den Gültigkeitsbereich des WITH vergrößern. Eine Prozedur, die den Datensatz einer Person in der einfachsten Form auf dem Bildschirm ausgibt, könnte mit einer solchen WITH-Anweisung so aussehen:

```
PROCEDURE  Datenauszug (Person : TypPersondatensatz);
BEGIN
  WriteLn ('==================================================');
  WITH Person DO
    BEGIN
      WriteLn ('Name des/der Beschäftigten: ':30, Name);
      WriteLn ('Vorname, Namenszusätze: ':30, Vorname);
      WriteLn ('Personalnummer: ':30, Personalnummer);
      WriteLn ('Gehaltsgruppe: ':30, Gehaltsgruppe);
```

```
      IF Geschlecht = weiblich
        THEN WriteLn ('Geschlecht: ':30, 'w')
        ELSE WriteLn ('Geschlecht: ':30, 'm');
                                {Und so weiter}
    END;
  WriteLn  ('==================================================');
END (* von Datenauszug *);
```

Speichern in Textdateien

Um die Beispiele zum *TypPersondatensatz* praktisch auszuprobieren, schreiben Sie sich zunächst eine Prozedur *Erfassung*. Damit Sie nicht bei jedem Programmlauf alle Daten neu eintippen müssen, speichern Sie die Daten in Textdateien ab und lesen Sie sie bei Bedarf von dort wieder ein. So wie in der Prozedur *Datenauszug* die einzelnen Angaben zu jeder *Person* auf den Bildschirm geschrieben werden, können sie auch in eine Textdatei auf Diskette geschickt werden. Zu beachten sind eventuelle Unterschiede beim Öffnen und Schließen von Dateien, falls ein anderer Pascal-Dialekt als Turbo-Pascal benutzt wird (es sei *Normalstring* wieder als STRING [80] vereinbart):

```
PROCEDURE  Speichern
  (Person    : TypPersondatensatz;
  Dateiname : Normalstring);

VAR
  Datei : Text;

BEGIN
  Assign (Datei, Dateiname);
  ReWrite (Datei);
  WITH Person DO
    BEGIN
      WriteLn (Datei, Name);
      WriteLn (Datei, Vorname);
      WriteLn (Datei, Personalnummer);
      WriteLn (Datei, Gehaltsgruppe);
      IF Geschlecht = weiblich
        THEN
          WriteLn (Datei, 'w')
        ELSE
          WriteLn (Datei, 'm');
                                {Und so weiter}
    END;
  Close (Datei);
END (* von Speichern *);
```

Die Daten werden zeilenweise in der Datei abgelegt. Nach einer Dateieröffnung mit Reset erhalten Sie sie mit ReadLn-Anweisungen zurück.

Das Beispiel für den Bücherfreund

Ein anderes Beispiel: Vielleicht stapeln sich bei Ihnen die gelesenen und ungelesenen Bücher, und Sie hegen die Hoffnung, mittels EDV die Übersicht zurückzugewinnen. Bei aller Mühsal des späteren Programmierens und auch der Programmnutzung liegt die schwierigste, verantwortungsträchtigste und folgenreichste Aufgabe am Anfang der Arbeit. Zuallererst müssen Sie sich überlegen, welche Daten benötigt werden, damit das Programm irgendwann Auskünfte auf Sie interessierende Fragen parat halten kann. Gesucht wird ein Modell für Bücher, das, mit spärlichen Angaben ausgerüstet, in der Mikrowelt des Computers die Originale vertreten muß. Die Typvereinbarung kann beispielsweise so aussehen:

```
TYPE
  TypKategorie = (Sachbuch, Belletristik, Lexikon);
  TypBuch      =
    RECORD
      AutorIn                   :  STRING [50];
      Titel                     :  STRING [60];
      Untertitel                :  STRING [60];
      Verlag                    :  STRING [30];
      Erscheinungsort           :  STRING [30];
      Erscheinungsjahr          :  STRING [4];
      JahrErstveroeffentlichung :  STRING [4];
      Schlagworte               :  STRING [240];
      Gelesen                   :  Boolean;
      Kategorie                 :  TypKategorie;
    END;
```

Erwägungen zum Datenmodell

Nur einen kleinen Ausschnitt beispielhafter Daten gibt die Tabelle in Bild 14-4 wieder. Vielleicht sollen die Bücher je nach Größe auf Borde mit verschiedenen Abständen gestellt werden? Wenn das DV-System Sie über die Möglichkeiten hierfür informieren soll, benötigen Sie zusätzlich die Komponente:

```
Hoehe : Real;                {Buchrücken in cm}
```

Möchten Sie auch ausrechnen lassen, wieviel Meter Regal Sie noch errichten müssen? Mit folgender Datenart kann der Computer das leicht erledigen:

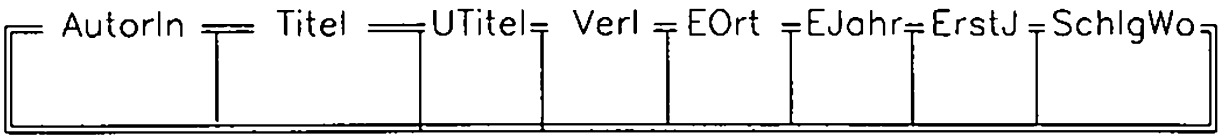

Abbildung 14-3: Schemaskizze von TypBuch

AutorIn	Titel	EJahr
Weizenbaum, Joseph	Die Macht der Computer und die	1977
Wirth, Niklaus	Algorithmen und Datenstrukturen	1983
Wirth, Niklaus	Programming in Modula-2	1982
Eurich, Claus	Computerkinder	1985
Zehnder, Carl August	Informationssysteme und Datenba	1985
Jackson, Michael A.	Grundsätze des Programmentwurfs	1979
Heinig, Sabine/ Lenz, Ilse	Schöne Neue Frauen-Welt, Comput	1988
Hansen, Hans Robert	Wirtschaftsinformatik I	1986
Faulstich/ Faulstich-Wieland	Computer-Kultur	1988
Röder, Rupert	Pascal	1991

Abbildung 14-4: Beispieldaten für TypBuch

```
Breite : Real;              {Buchrücken in cm}
```

Um für Buchempfehlungen auch die Preise berücksichtigen zu können, benötigen Sie die Attribute:

```
Preis                       :  Real;
PreisDatum                  :  STRING [8];
                            {Wann der Preis galt.}
```

Gerade bei Computerliteratur ist aus dem Titel nicht unbedingt ersichtlich, in welcher Sprache das Buch geschrieben ist. Ein weiterer Hinweis kann davor schützen, sich mit einem Text in einer nicht beherrschten Fremdsprache zu plagen:

```
Sprache                 :  STRING [10];
```

Das von Ihnen gewählte Datenmodell bestimmt, über welche Daten Sie verfügen und welche Informationen Sie der entsprechenden Datenbank entnehmen können.

Die Prozeduren zum Aufnehmen, Wiedergeben, Speichern und Lesen der Buch-Daten gleichen im Aufbau den entsprechenden Funktionen für eine Personaldatenverwaltung. Standardprozeduren wie die genannten müssen im Grunde bloß einmal entworfen und für die jeweilige Anwendung nur durch Einsetzen der entsprechenden Texte und Komponentennamen angepaßt werden.

Die Sprachen der 4. Generation

In der DV-Entwicklung werden heute oft Programmgeneratoren oder Programmiersprachen der 4. Generation verwendet. Ein wesentlicher Bestandteil solcher Programmiersysteme sind fertige Standardprozeduren für die komfortable Ein- und Ausgabe von Datensätzen. Um jedoch auf die Anwendungssituation zugeschnittene Dialoge zu entwerfen und Masken und Menüs genau auf den Punkt zu bringen, ist es oft besser, anstelle von vorfabrizierten doch eigene in Pascal oder einer anderen Programmiersprache der 3. Generation geschriebene Prozeduren einzusetzen.

Bis jetzt war nur von einem beziehungsweise einzelnen Objekten vom *TypBuch* die Rede. Wie ist das ganze *BuecherRegal* oder wie wäre eine komplette *Personalliste* in einem Programm unterzubringen und anzusprechen?

ARRAYs

Wenn eine Gruppe gleichartiger Daten gemeinsam die Grundlage für einen Verarbeitungsvorgang bilden soll (etwa um eine alphabetisch sortierte Liste zu erstellen oder rasch nach einem Begriff zu suchen), müssen die Daten parallel im internen Speicher gehalten werden. Aus vielen identisch aufgebauten Einzelgrößen kann für diesen Zweck mittels Reihung eine Super-Variable, ein ARRAY, gebildet werden. Während die Datenstrukturierung beim RECORD mehrere *unterschiedliche* Attribute unter einen Hut bringt, fügt das ARRAY (oder die „Reihe") eine Vielzahl *gleichartiger* Komponenten zu einer Einheit zusammen.

Beispiel

Im Beispiel von oben würde die Variable für ein ARRAY, bestehend aus zum Beispiel maximal 50 Einzelpersonen, so deklariert werden:

```
VAR
  Personalliste : ARRAY [1..50] OF  TypPersonDatensatz;
```

In dieser Form handelt es sich um die implizite Vereinbarung eines ARRAY-Typs. *TypPersondatensatz* spielt hier die Rolle des Komponententyps des ARRAYs. Die Angabe 1..50 in eckigen Klammern besagt, daß 50 Datensätze angelegt werden, die mit den ganzen Zahlen von 1 bis 50 durchnumeriert sind.

Index

Statt von Numerierung wird in Pascal allgemeiner von einer Indizierung durch den Index beziehungsweise den Indextyp gesprochen. 1..50 kann nämlich formal als Teilbereichstyp von Integer aufgefaßt werden. Der Indextyp wird in eckigen Klammern geschrieben wie schon die SETs. Der Zusammenhang läßt aber keine Verwechslung zu. Als Indextyp ist jeder Aufzählungstyp oder ein Teilbereich zugelassen.

Ansprechen der ARRAY-Komponente

Praktisch erhalten Sie eine Reihe von 50 gleichartigen Variablen, die mit ihrer Nummer (beziehungsweise ihrem sonstigen Index) zu identifizieren sind. Um eine bestimmte davon anzusprechen, schreiben Sie einfach die Nummer in eckigen Klammern zum Variablennamen hinzu. Etwa

```
Personalliste [6]
```

bedeutet nicht eine sechste Liste, sondern bezeichnet die Position Nr. 6 auf der Liste, das heißt die Daten der sechsten Person.

Explizite Deklaration

Selbstverständlich ist auch eine explizite Typvereinbarung mit anschließender Variablendeklaration möglich:

```
TYPE
  TypPersonalliste : ARRAY [1..50] OF TypPersonDatensatz;
VAR
  Personalliste     : TypPersonalliste;
```

Abarbeiten mit der FOR-Schleife

In charakteristischer und außerordentlich praktischer Form können häufig die Daten eines ARRAYs mit einer FOR-Schleife abgearbeitet werden. FOR-Schleifen sind ja dazu bestimmt, die Zahlen oder andere geordnete Wertefolgen zu durchlaufen. Zum Beispiel kann die oben angeführte Prozedur *Datenauszug* mit einer schlichten Anweisung die ganze Personalliste auf dem Bildschirm ausgeben (*i* sei als Integer deklariert):

```
FOR i := 1 TO 50 DO
  Datenauszug (Personalliste [i]);
```

Ganz ähnlich wurde bereits mit STRINGs gearbeitet. Intern sind STRINGs beinahe ARRAYs mit dem Komponententyp Char, nur daß Sie zusätzlich eine Längenangabe enthalten.

Computerisierung für das BuecherRegal

Auch für den Bücherfreund von oben soll nun eine Datenstruktur mit ARRAY aufgebaut werden. Damit die Fassungsgrenze der Regale nach oben mühelos erweitert werden kann, sei die obere Indexgrenze als Konstante vereinbart:

```
CONST
  K {apazitaet}   = 100;
TYPE
  TypKategorie    = (Sachbuch, Belletristik,
                     Lexikon);
  TypBuch         =
    RECORD
                              { ... }
    END;
  TypBuecherRegal = ARRAY [1..K] OF
                     TypBuch;
VAR
  BuecherRegal     : TypBuecherRegal;
```

Die komplette Struktur für eine umfangreiche Tabelle ist nun fertig – die Felder müssen nur noch mit den Daten gefüllt werden.

Sind die Daten im ARRAY, ist es nicht schwer, sie in eine Textdatei zu übertragen beziehungsweise wieder aus dieser zu holen. Zum Beispiel speichert die folgende Prozedur die ersten (von 1 bis *AnzahlSaetze*) Komponenten des ARRAYs auf die Datei 'BIBLIOT.HEK':

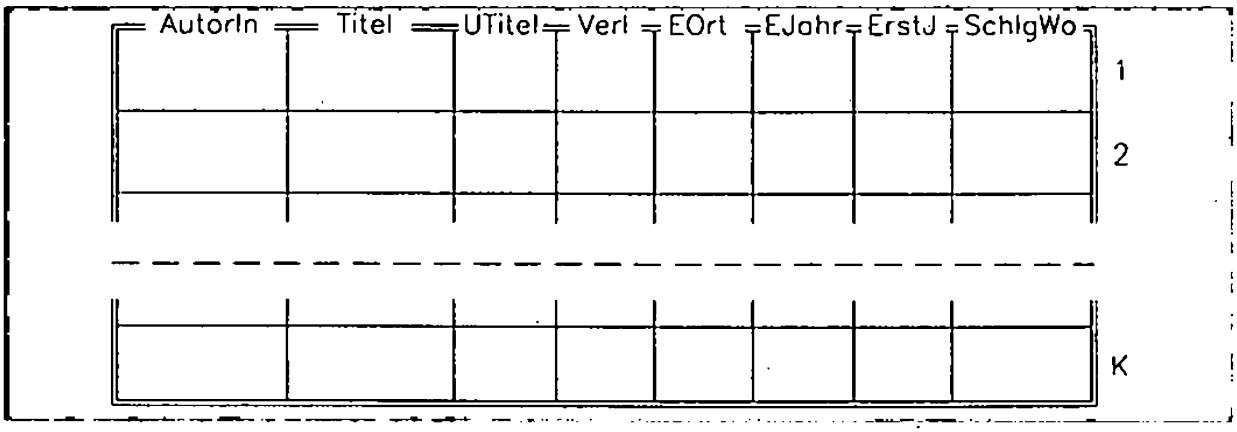

Abbildung 14-5: Tabelle

```
PROCEDURE Notiere (AnzahlSaetze: Integer);
VAR
  Datei : Text;
  i     : Integer;
BEGIN
  Assign (Datei, 'BIBLIOT.HEK');
  ReWrite (Datei);
  FOR i := 1 TO AnzahlSaetze DO
    Speichern (Datei,BuecherRegal [i]);
  Close (Datei);
END;
```

Dabei sei angenommen, daß zuvor folgende Prozedur *Speichern* mit *Datei* als Variablenparameter und einem Werteparameter vom *TypBuch* deklariert wurde:

```
PROCEDURE Speichern (VAR Datei: Text; Buch: TypBuch);
BEGIN
  WITH Buch DO
    BEGIN
      WriteLn (Datei, AutorIn);
      WriteLn (Datei, Titel);
      { und so weiter }
    END;
END;
```

Der Abstieg zu den elementaren Werten

Für den Komponententyp bei einem ARRAY gibt es ebensowenig Einschränkungen wie für den eines RECORDs. Aus jedem elementaren, aber auch aus jedem schon in sich strukturierten Datentyp darf eine Reihe gebildet werden. In allen bisherigen Beispielen war das Reihenelement bereits selbst ein RECORD. Machen Sie sich klar, in welchen Stufen Sie bei solchen geschachtelten Konstruktionen an die einfachen Werte herankommen:

```
BuecherRegal [11]
```

stellt ein Objekt vom *TypBuch* dar und enthält alle Angaben zum elften Buch im Regal. Für den *AutorIn*-Namen, den *Titel* oder das *Erscheinungsjahr* muß auf die betreffenden RECORD-Komponenten zugegriffen werden:

```
BuecherRegal [11]. AutorIn
BuecherRegal [11]. Titel
BuecherRegal [11]. Erscheinungsjahr
```

Je länger die RECORD-Bezeichnung wird, desto mehr kann sich der Gebrauch eines WITH lohnen. Allerdings

kommt es gar so nicht häufig vor, daß die Datenstruktur ganz durchbuchstabiert werden muß. Meistens können komplette Teilstrukturen (wie zuletzt *TypBuch* in der Prozedur *Speichern*) durch einen Prozedurparameter abgedeckt werden.

Geschachtelte Datentypen

Das Datenbild einer Kurve

Natürlich gibt es auch ganz schlichte ARRAYs, die nichts als eine Reihe einfacher Zahlengrößen aufnehmen. Für Zwecke einer Analyse internationaler Wirtschaftsbeziehungen interessieren Sie sich vielleicht für den Verlauf der Fieberkurve des Dollarkurses. Eine Datenstruktur, zur Aufnahme der Werte eines bestimmten Zeitraums, kann wie folgt aufgebaut werden:

```
CONST
  Zeitspanne     =  37;     {Tage}
TYPE
  TypKursreihe   = ARRAY [1..Zeitspanne]
                   OF Real;
VAR
  Dollarkurve    : TypKursreihe;
```

Eine Prozedur zum *Zeichnen* der *Kurve* kann (wenn die Daten schon vorhanden sind) problemlos geschrieben

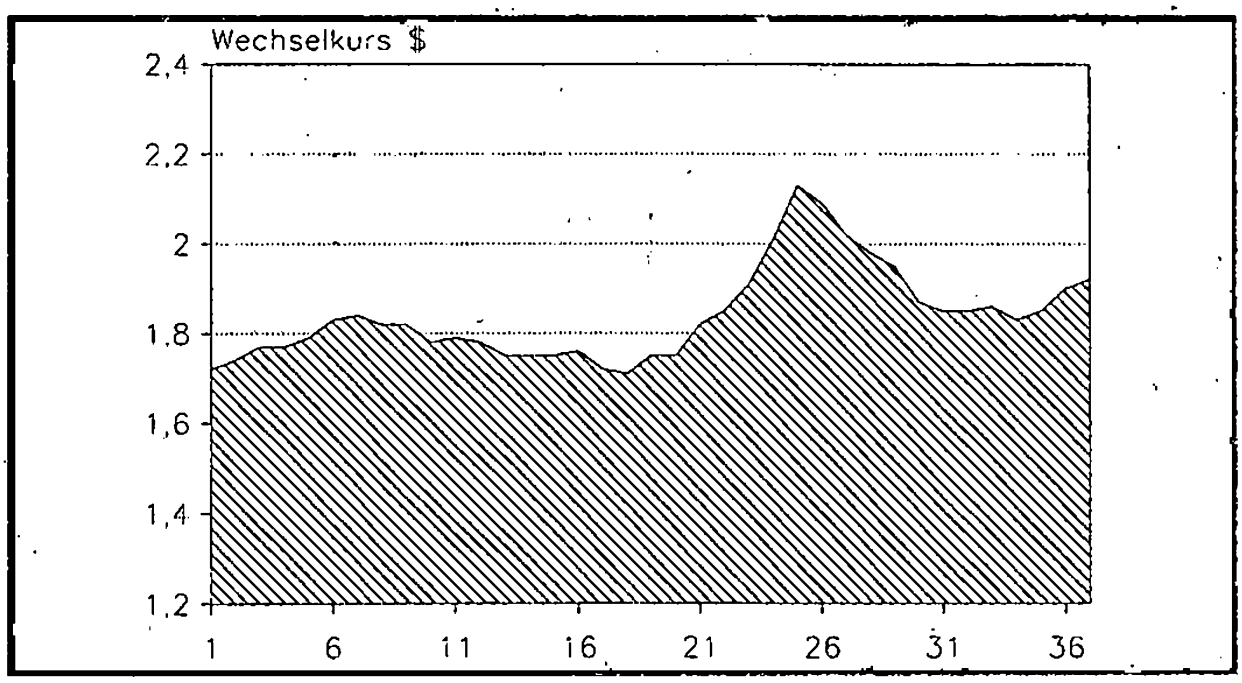

Abbildung 14-6: Eine Wechselkurs-Kurve

werden. Um Kurve und Koordinatenkreuz auf dem Bildschirm zu plazieren, werden zuvor globale Konstanten benötigt:

```
CONST
  KoordinatenAnfangszeile   = 24;
  KoordinatenAnfangsspalte =  5;
  Vergroesserungsfaktor     = 10;

PROCEDURE Zeichnen (Kurve: TypKursreihe);
VAR
  Wert            : Real;
  HoeheAufSchirm : Integer:
  Tag             : Integer;
BEGIN

  FOR Tag := 1 TO Zeitspanne DO
    BEGIN
      HoeheAufSchirm := round (Vergroesserungsfaktor
                               * Kurve [Tag]);
      GoToXY (Tag + KoordinatenAnfangsspalte,
              KoordinatenAnfangszeile - HoeheAufSchirm);
      Write ('*');
    END;
END;
```

Mehr als ein einfaches ARRAY

Diese Lösung weist noch einen erheblichen Schönheitsfehler auf: Im Programm muß, wenn die Daten sich erst im Laufe der Zeit ansammeln, täglich die Konstante *Zeitspanne* geändert werden. Eigentlich sollten die ARRAY-Grenzen von vorneherein so gewählt werden, daß Daten von einem größeren Zeitraum, etwa einem Jahr, als *MaximaleZeitspanne*, aufgenommen werden können. Man muß sich dann allerdings getrennt merken, wie viele Werte schon vorhanden sind (bis zu welchem Tag die Kurve zu zeichnen ist). Dafür könnte eine isolierte Variable „Zeitspanne, der schon durchgeführten Beobachtung" definiert werden. Weit eleganter ist es allerdings, wenn zur Beschreibung einer Kurve nicht nur die Werte, sondern auch die Zeitspanne, für die Werte vorliegen, als zugehörig erkannt wird. In Pascal gesprochen: *Kurve* sollte als ein RECORD mit zwei Komponenten – einem ARRAY OF Real und einer Integer-Größe *Zeitspanne* – aufgefaßt werden. Die Datenstruktur sieht dann so aus:

```
CONST
  MaximaleZeitspanne = 365;
                          {Tage}
TYPE
  TypKursreihe        = ARRAY [1..MaximaleZeitspanne] OF Real;
  TypKurve            =
    RECORD
       Zeitspanne : Integer;
       Kursreihe  : TypKursreihe;
    END;
VAR
  Dollarkurve         : TypKurve;
```

Die Prozedur muß nur geringfügig abgeändert werden. Bevor die notwendige Modifikation gezeigt wird, soll auf ein Problem mit den Typangaben eingegangen werden, das vor allem bei RECORD-Komponenten interessant wird:

Teilbereichstypen verwenden?

In der Pascal-Literatur findet sich oft die Empfehlung, in einer RECORD-Vereinbarung bei Integer- oder Char-Komponenten als Typ jeweils einen genau passenden Teilbereich von Integer beziehungsweise Char zu nehmen (ähnlich wie bei STRINGs die jeweilige Maximallänge genannt wird). Weil zum Beispiel *Zeitspanne* höchstens gleich *MaximaleZeitspanne* sein darf, könnte deklariert werden:

```
TypKurve        =
  RECORD
    Zeitspanne : 1..MaximaleZeitspanne;
    Kursreihe  : TypKursreihe;
END;
```

Dann würde das Pascal-System bei entsprechender Voreinstellung dafür sorgen, daß kein unzulässiger Wert an die Variable zugewiesen wird. Aber wie schon in Kapitel 12 begründet, sollte lieber mit eigenen Prozeduren wie zum Beispiel Eingabekontrollen verhindert werden, daß Variablen den ihnen zustehenden Bereich überschreiten. Zu den geschilderten Unzulänglichkeiten der Pascal-Typenkontrolle für Zwecke der Bereichsprüfung kommt hinzu, daß bei jeder Prozedur die Parameter strikt typgebunden sind. Würde mit Teilbereichstypen gearbeitet, müßten zum Beispiel für jeden Teilbereich von Integer eigene Kontroll-, Bearbeitungs- und Ausgabeprozeduren definiert werden.

Es kann allerdings sinnvoll sein, für Variablen desselben formalen Typs, deren Werte aber ganz unterschiedliche Größen repräsentieren, die Typnamen zu trennen. So könnte definiert werden:

```
TypZeitachse              =  Integer;
TypBildschirmkoordinaten  =  Integer;
```

Nun kann *Zeitspanne* als vom *TypZeitachse*, dagegen *HoeheAufSchirm* als vom *TypBildschirmkoordinaten* ausgewiesen werden.

Die neue Prozedur Zeichnen

Zum neuen Modell für die *Kurve*: Was in der ursprünglichen Datenstruktur *Kurve [Tag]* hieß, muß nach der Erweiterung jetzt mit *Kurve.Kursreihe [Tag]* angesprochen werden. Ebenso ist *Zeitspanne* zu qualifizieren. Die Prozedur hat dann folgende Gestalt:

```
PROCEDURE Zeichnen (Kurve: TypKursreihe);
VAR
  Wert            : Real;
  HoeheAufSchirm  : Integer:
  Tag             : Integer;
BEGIN
  FOR Tag := 1 TO Kurve.Zeitspanne DO
    BEGIN
      HoeheAufSchirm := round ( Vergroesserungsfaktor *
                                Kurve.Kursreihe [Tag] );
      GoToXY (Tag + KoordinatenAnfangsspalte,
              KoordinatenAnfangszeile - HoeheAufSchirm);
      Write ('*');
    END;
END;
```

Die Überlegungen bezüglich des Modells für eine Kurve sind auf die anderen Beispiele zu übertragen, auch wenn die Datenstrukturen dabei immer weiter wachsen.

Tief geschachtelte Datentypen

Wie schon erwähnt, können ARRAYs wie RECORDs in Pascal praktisch Komponenten jedes Typs besitzen. Das eröffnet die sehr sinnvolle Möglichkeit, Datentypen in mehreren Stufen zu schachteln. Man bildet RECORDs von ARRAYs, deren Elemente wiederum RECORDs von allem möglichem sein können (dabei gilt es, die Übersicht zu behalten).

Hierzu noch einmal das Beispiel des Bücherfreundes: Beim *BuecherRegal* sollte zwischen maximaler Aufnahme-

kapazität und aktueller Belegungszahl unterschieden werden. Die verbesserte Datenstruktur lautet:

```
CONST
  Kapazitaet       = 100;
TYPE
  TypKategorie     = (Sachbuch, Belletristik, Lexikon);
  TypBuch          =
    RECORD
                          { ... }
    END;
  TypBuecherRegal = ARRAY [1..Kapazitaet] OF TypBuch;
  TypBibliothek    =
    RECORD
       BuecherRegal   : TypBuecherRegal;
       Belegungszahl  : Integer;
    END;
VAR
  Bibliothek       : TypBibliothek;
```

Wer schrieb laut dieser Datenstruktur das letzte ins Regal aufgenommene Buch? Es kann niemand anders sein als:

```
Bibliothek . BuecherRegal [Belegungszahl] . AutorIn
```

Das durchaus ernsthafte Spiel läßt sich noch weiter treiben. Eine *Bibliothek* umfaßt meist eine ganze Reihe von *BuecherRegalen*, beziehungsweise sie ist nach Sachgebieten gegliedert. Versuchen Sie doch einmal, eine Datenstruktur für diese Situation aufzustellen. Spätestens, wenn die Bildschirmzeile nicht mehr reicht, um die Komponenten anzusprechen, können Sie auch Wertzuweisungen zwischenschalten. Mit einer Hilfsvariable *Buch* vom *Buchtyp* hätte nach Ausführung von:

```
Buch := Bibliothek.BuecherRegal [Belegungszahl];
```

die gewünschte Auskunft auch auf:

```
Buch.AutorIn
```

gestanden.

Jackson-Diagramme

Die Datenstrukturierung läßt sich in ähnlicher Form in hierarchisch übereinander angeordneten Kästchen darstellen wie die Prozeduren im Problemaufbaugraph der Top-down-Programmierung.

Das Diagramm deutet an, daß *Bibliothek* durch zwei Komponenten bestimmt ist, von denen *Buecherregal* sei-

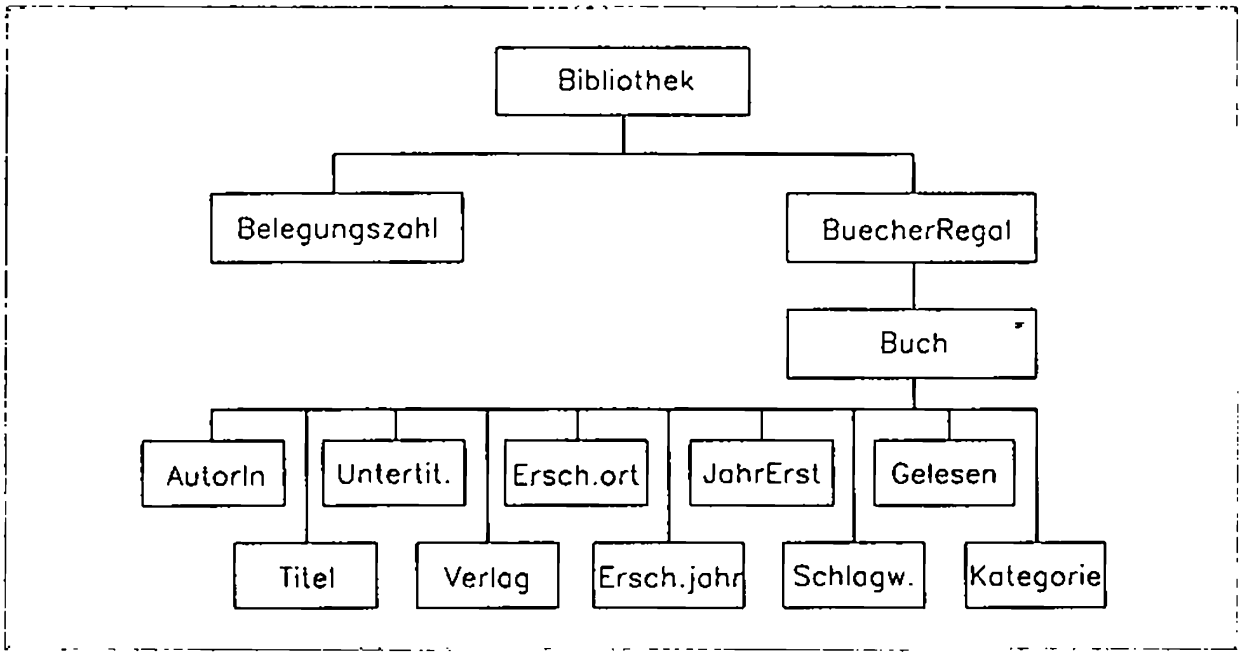

Abbildung 14-7: Ein Datenstruktur-Diagramm

nerseits aus vielen Exemplaren *Buch* besteht. Das Sternchen rechts oben im Kästchen signalisiert ein mögliches mehrfaches Vorkommen der entsprechenden Einheit. Die Komponenten eines jeden *Buch*-Exemplars nehmen die unterste Ebene ein. Ein derartiges Datenstruktur-Diagramm kann (beziehungsweise muß bei der sogenannten Programmiermethodik nach Jackson) als direkte Anleitung zum Top-down-Entwurf des zugehörigen Programms dienen: Insgesamt ist ein Programm zu schreiben, das sich mit der *Bibliothek* befaßt. Die Vielzahl der *Buch*-Exemplare spiegelt sich im Programm in einer Wiederholungsstruktur, welche in ihrem Innern jeweils ein *Buch* behandelt. Die Prozedur für das *Buch* untergliedert sich ihrerseits in Unterprozeduren für die einzelnen Datenfelder von *AutorIn* bis *Gelesen*, die eine nach der anderen in einer Programmsequenz aufgerufen werden.

Variante RECORDs

Die dritte der grundlegenden Programmstrukturen – die Entscheidungsstruktur – käme zum Zug, wenn je nach Kategorie des Buchs unterschiedliche Datenfelder erforderlich wären. Die entsprechende Pascal-Datenstruktur ist der variante Record, für den hier nur ein kurzes Beispiel stehen soll:

```
TypBuch      =
  RECORD
    AutorIn          : STRING [50];
    Titel            : STRING [60];
                             { ... }
    Gelesen          : Boolean;
    CASE Kategorie : TypKategorie  OF
      Sachbuch       : (Fachgebiet : STRING [30];
                        Zielgruppe : STRING [20]);
      Belletristik : (Genre       : STRING [40];
                        Vorbilder  : STRING [150]);
      Lexikon        : (Niveau     : STRING [20]);
END;
```

Es werden am Schluß der RECORD-Definition in einem CASE-Abschnitt mehrere Fälle aufgelistet. Je nach Wert des Etikettenfelds *Kategorie* sind unterschiedliche Teil-Datensätze zu aktivieren (im Gesamt-Datensatz wird Raum für die längste der verschiedenen Versionen freigehalten). Nach der folgenden Zuweisung (wobei *Buch* vom *TypBuch* ist) kann mit den Komponenten *Buch.Genre* oder *Buch.Vorbilder* gearbeitet werden:

```
Buch.Kategorie := Belletristik;
```

Falls *Buch.Kategorie* gleich *Sachbuch* ist, gelten dagegen die Komponenten *Buch.Fachgebiet* und *Buch.Zielgruppe*, wenn das Etikett *Lexikon* enthält, ist *Buch.Niveau* aktiv. Im Jackson-Diagramm erscheinen alternative Komponenten mit einem Kringel rechts oben im Kästchen (Abb. 14-8).

Speicherbedarf der Daten

Bei großen ARRAYs stößt man leicht an die Grenzen des Speichers des Computers beziehungsweise an die

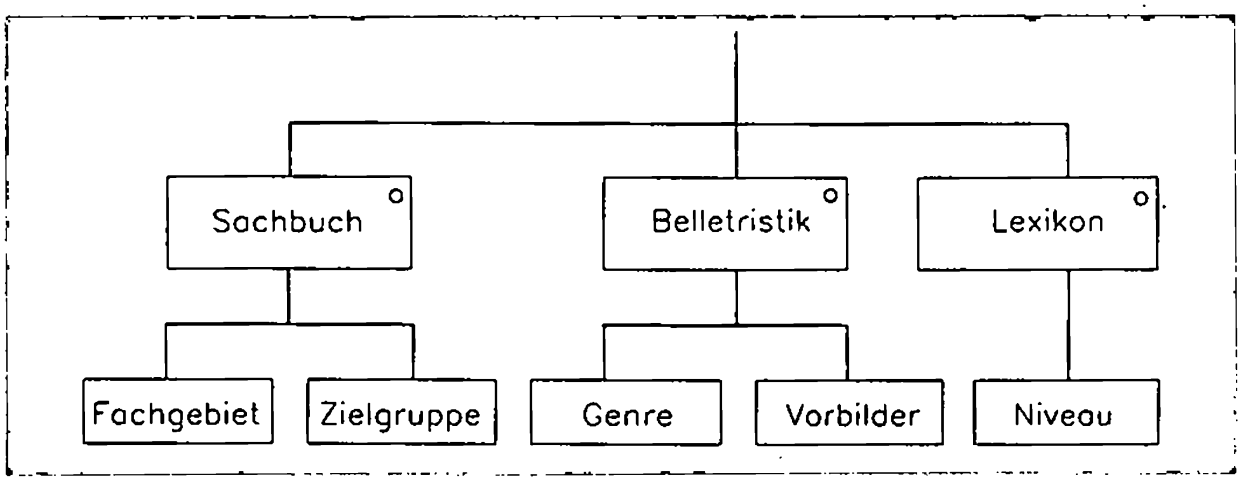

Abbildung 14-8: Datenstruktur-Diagramm mit variantem RECORD

Grenzen des für statische Variablen zugänglichen Speicherabschnitts (in Turbo-Pascal je Variable maximal 64 kByte). Es lohnt sich daher, beim Definieren von umfangreichen ARRAYs mit großen Komponenten von Zeit zu Zeit deren Speicherbedarf zu überschlagen. Schon in der ersten, noch nicht erweiterten Form wird je Datensatz vom *TypBuch* zirka ein halbes KByte Speicher benötigt. Ein ARRAY, das die Daten von 500 Bänden aufnehmen soll, würde 250 KByte in Beschlag nehmen. Wenn die Datenbestände derartig umfangreich werden, können sie portionsweise in eine oder mehreren Dateien ausgelagert werden.

Die Datenstruktur FILE OF für den Direktzugriff auf Datensätze

Die oben geschilderte Möglichkeit, Daten aus RECORDs in Textdateien zu schreiben, hat den Vorteil, daß sie bei allen Pascal-Dialekten ganz ähnlich funktioniert. Außerdem werden die Daten zeichenweise im ASCII-Format notiert, so daß zahlreiche andere Programme darauf zugreifen können. Beliebige Dateneinheiten können aber in Pascal auch (in einem von der jeweiligen Pascal-Implementation abhängigen Format) geblockt in einer Datei, einem FILE, gespeichert werden. Eine allgemeine Datei besteht in Pascal aus einer theoretisch unbeschränkten, praktisch nur durch die Kapazität des Mediums begrenzten Zahl von Datenobjekten desselben Typs. Die Struktur des FILE ähnelt der des ARRAY, nur daß letzterer eine feste Anzahl von Objekten eines Typs umfaßt und im internen Speicher steckt, während das FILE extern bleibt.

Deklaration

Zum Beispiel kann der Bücherfreund sich folgende Dateistruktur zulegen:

```
TYPE
  TypKategorie     = (Sachbuch, Belletristik, Lexikon);
  TypBuch          =
    RECORD
                              { ... }
    END;
  TypBuecherDatei = FILE OF TypBuch;

VAR
  BuecherDatei     : TypBuecherDatei;
```

In der Typbeschreibung steht hinter FILE OF die Angabe des Komponententyps der Datei (in diesem Fall *TypBuch*). Diese Deklarationen schließen nicht aus, daß zusätzlich Objekte vom *TypBuecherRegal* beziehungsweise *TypBibliothek* vereinbart werden, um eine gewisse Anzahl von Buchdatensätzen zum schnellen Zugriff im internen Speicher zu halten.

Arbeiten mit strukturierten Dateien

Das Schreiben und Lesen auch von beliebig großen Datenobjekten in strukturierte Dateien gestaltet sich sehr leicht. Genau wie eine Textdatei, müssen auch andere FILEs zunächst eröffnet werden. Im folgenden sei zugrundegelegt, daß auf der String-Variablen *Dateiname* der nach den Regeln des Betriebssystems gebildete Name einer Datei steht. Falls die Datei existiert, wird sie in Turbo-Pascal eröffnet mit

```
Assign  (BuecherDatei, Dateiname);
Reset   (BuecherDatei);
```

Die Neuanlage der Datei erfolgt mit den Anweisungen

```
Assign  (BuecherDatei, Dateiname);
ReWrite (BuecherDatei);
```

Um jeweils ein komplettes Glied des FILEs vom Speichermedium in eine Variable *Buch* vom *TypBuch* einzulesen beziehungsweise von der Variablen auf die Datei zu übertragen, wird es in Turbo-Pascal einfach hineingeschrieben beziehungsweise herausgelesen:

```
Write (BuecherDatei, Buch);
Read  (BuecherDatei, Buch);
```

In Standard-Pascal dürfen Write und Read nur bei Textdateien angewendet werden. Für allgemeine Dateien sind sogenannte Get- beziehungsweise Put-Operationen definiert, die ein Schreib-/Lesefenster (analog zum physikalischen Schreib/Lesekopf) in der Datei verschieben. Am Ende der Arbeit muß die Datei wieder geschlossen werden:

```
Close (Buecherdatei);
```

Ebenso wie bei Textdateien kann mit *eof (Buecherdatei)* abgefragt werden, wann alle Datensätze durchlaufen sind. Falls man es für sinnvoll erachtet, kann man anstelle der *Buecherdatei* auch eine *BuecherRegalDatei* oder eine *BibliothekenDatei* anlegen.

```
TYPE
  TypBibliothekenDatei =  FILE OF TypBibliothek;
VAR
  BibliothekenDatei    :  TypBibliothekenDatei;
```

Bei dieser Deklaration schreiben Sie nachher mit einer einzigen Lese- beziehungsweise Schreibanweisung eine ganze Bibliothek zwischen Diskette und internem Speicher hin- und her.

Möglichkeit des Direktzugriffs

Eine strukturierte Datei erspart das Schreiben von eigenen kleinen Prozeduren für die Ein- und Ausgabe von Datensätzen. Interessant wird diese Speicherungsform aber vor allem durch ihre Eigenschaft, aus lauter gleichartigen Komponenten zu bestehen. Aufgrund der festen Größe der Komponenten ist es im Prinzip möglich, daß Pascal-System und Betriebssystem ermitteln, an welchem Platz im Speicher die soundsovielte Komponente stehen muß. Sofern der verwendete externe Speicher (wie bei Disketten und Platten der Fall) das gezielte Anspringen bestimmter Stellen zuläßt (Direktzugriff), kann ein bestimmtes Dateiglied direkt angesprochen werden. Ohne erst sämtliche vorherigen Daten abzusuchen, können gezielt Feldinhalte gelesen und verändert werden. Alle Pascal-Dialekte für Mikrocomputer sehen solche Möglichkeiten vor; das ursprünglich für Großrechner alten Stils mit Magnetbandgeräten als Speichereinheiten gedachte Standard-Pascal allerdings nicht. In Turbo-Pascal läßt sich der direkte Zugriff auf eine bestimmte Dateikomponente mit der Seek-Anweisung („suche") zustande bringen. Zum Beispiel setzt die folgende Anweisung den Dateizeiger auf das siebte Glied der Datei (die Komponenten werden mit 0 beginnend gezählt):

Suchen

```
Seek (BuecherDatei,  6)
```

Eine folgende Lese- oder Schreiboperation bezieht sich dann auf die zuvor mit Seek angesteuerte Komponente.

Auf eine Spezialaufgabe der Dateiverwaltung sei noch kurz eingegangen: das Sortieren von Daten in alphabetischer (beziehungsweise alphanumerischer) oder einer anderen Reihenfolge.

Sortieren

Ein Sortierverfahren für den Computer wird auch ein Sortieralgorithmus genannt. Der traditionsreiche Begriff

des Algorithmus (nach dem latinisierten Namen des persischen Rechenbuchautors Al-Chwarizmi aus dem 8. Jahrhundert) ist der Mathematik entlehnt und bezeichnet jedes mechanisch nach genauen Regeln ausführbare Lösungsverfahren für eine bestimmte Klasse von Aufgaben. Vorgestellt wird eines der einfachsten, aber für kleine Datenumfänge durchaus effektives Sortierverfahren.

Der Grundgedanke beim Sortieren *durch direkte Auswahl* lautet, daß zunächst das ganz zuvorderst gehörende (je nach Sortierung das kleinste, größte oder alphabetisch erste usw.) Datenelement gesucht und an den Anfang der Reihe plaziert wird. Der neue Kopf der Reihe ist damit schon fertig, auf den Reihenrest wird das Verfahren erneut angewendet. Das dort zuvorderst gehörende Element wird gesucht, gefunden und auf den vorderen Platz umgesetzt. Eine Prozedur, welche die Datenreihe innerhalb der Datenstruktur vom *TypBibliothek* alphabetisch nach dem Datenfeld *AutorIn* ordnet, könnte etwa wie nachfolgend aussehen. Vorauszusetzen ist, daß das Datenfeld jetzt mit dem explizit definierten *TypNamenString* statt wie oben mit STRING [60] beschrieben wird. Die alphabetische Reihenfolge kann wie die Ordnung bei Zahlen abgefragt werden (allerdings nach der ASCII-Tabelle, das heißt die deutschen Sonderzeichen kommen an der falschen Stelle und in falscher Anordnung):

```
PROCEDURE Sortieren (VAR Bibliothek : TypBibliothek);
VAR
  AnfangsnummerRestreihe,
  GefundeneNummer          : Integer;
PROCEDURE SucheNachVorneGehoerendesElement
        (VAR GesuchteNummer      : Integer;
         AnfangsnummerRestreihe : Integer);
VAR
  i, Nummer : Integer;
  Vorschlag : TypNamenString;
                          {Vorschlag für nach vorne
                          gehörendes Datenelement.}
BEGIN
  Vorschlag  := BuecherRegal [AnfangsnummerRestreihe]. AutorIn
  Nummer     := AnfangsnummerRestreihe;
  FOR i:= AnfangsnummerRestreihe + 1 TO Belegungszahl DO

              {Wenn AutorIn des i-ten Buchs vor dem alten Vorschlag
              im Alphabet steht, so merke i auf Nummer und halte
              AutorIn als neuen Vorschlag fest.}
```

```
    IF BuecherRegal [i].AutorIn < Vorschlag
      THEN
        BEGIN
           Nummer := i;
            Vorschlag := BuecherRegal [i].AutorIn;
        END;
             {Die Nummer, die schließlich stehen geblieben ist,
             bezeichnet das Buch, das alphabetisch nach vorne
             gehört.}

  GesuchteNummer :=  Nummer;
END (* von SucheNachVorneGehoerendesElement *);

PROCEDURE Platztausch (VAR Element1, Element2 :TypBuch);
VAR
  Element : TypBuch;
BEGIN
  Element  := Element2;    {Merke Element2 auf Element}
  Element2 := Element1;    {Überschreibe Element2 mit Element1}
  Element1 := Element;     {Setze Element, also das alte Element2,
                           auf Element1}
END (* von Platztausch *);

BEGIN
  WITH Bibliothek DO
    FOR AnfangsnummerRestreihe := 1 TO Belegungszahl - 1 DO
      BEGIN
        SucheNachVorneGehoerendesElement
          (GefundeneNummer, AnfangsnummerRestreihe);
        IF GefundeneNummer <> AnfangsnummerRestreihe
          THEN
             {Das alphabetisch zuvorderst gehörende Buch tauscht
             seinen Platz mit dem in der Restreihe bislang vorne
             stehenden.}
            Platztausch
              (BuecherRegal [AnfangsnummerRestreihe],
              BuecherRegal [NummerDesAlphabetischVorrangigen]);
      END;
END (* von Sortieren *);
```

Um eine Datei neu zu ordnen, können Sie übrigens auch auf das Betriebssystem zurückgreifen. In MS-DOS gibt es zum Beispiel den Befehl SORT zur alphabetischen Sortierung

Komplexe Objekte finden sich in fast jedem Zusammenhang, auch in relativ kleinen und überschaubaren Programmen. Zu dem Thema nun noch einige Beispiele.

Weitere Beispiele

Wie erwähnt, kann als Indextyp, mit dem ein ARRAY numeriert wird, jeder Aufzählungstyp beziehungsweise ein Teilbereich Verwendung finden.

ARRAY mit negativen Zahlen im Index

Etwa beim Beispiel der Kurve: Möchten Sie Daten vor dem Stichtag zusätzlich erfassen, können Sie auch einfach ein ARRAY bilden, dessen Index in den negativen Zahlen anfängt. Eine mögliche Datenstruktur wäre dann:

```
CONST
  MaximaleZeitspanne = 365; {Tage}
TYPE
  TypKursreihe       = ARRAY [-30..MaximaleZeitspanne] OF Real;
    TypKurve         =
      RECORD
         Zeitspanne : Integer;
         Kursreihe  : TypKursreihe;
      END;
```

Anstelle von -30 kann natürlich auch eine Konstante, etwa *Vorlaufspanne*, definiert oder gleich ein Typname für den gewünschten Bereich gebildet werden:

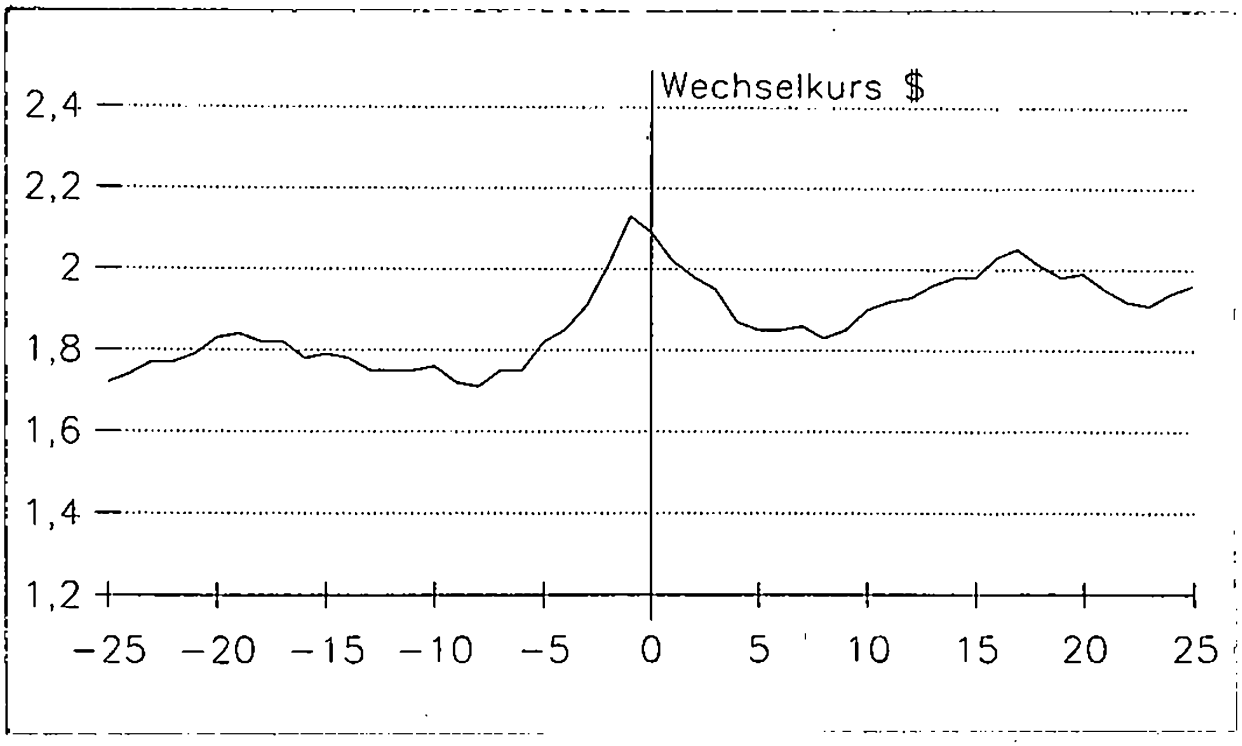

Abbildung 14-9: Kurve vor 0 beginnend

```
CONST
  MaximaleZeitspanne  =  365; {Tage}
  Vorlaufspanne       =  -30; {Tage}

TYPE
  TypZeitachse        = Vorlaufspanne .. MaximaleZeitspanne;
  TypKursreihe        = ARRAY [TypZeitachse] OF Real;

                          { ... }
```

Andere Aufzählungstypen als Indices

Manchmal ist es äußerst praktisch, daß ein ARRAY nicht nur mit ganzen Zahlen, sondern auch mit anderen Aufzählungstypen indiziert werden kann (etwa beim Apfelzucht-Beispiel). Wenn für die Apfelsorten ein Verzeichnis mit Informationen über die einzelnen Apfelsorten geführt werden soll, so bietet sich die Strukturierung durch *ARRAY [TypApfelsorte] OF* als optimal an:

```
TYPE
  TypApfelsorte     =  (Jonathan, Klarapfel, Boskop, CoxOrange,
                        SommerCox, GoldenDelicious, GrannySmith,
                        Ontario, Renette);
  TypVerwendung     =  (Tafel, Kochen, Backen, Most);
  TypMonatsSet      =  SET OF 1..12;
  TypSortenInformation =
    RECORD
       Reifezeit    :  TypMonatsSet;
       Geschmack    :  STRING [30];
       Aussehen     :  STRING [40];
       Verwendung   :  TypVerwendung;
       Name         :  STRING [50];
       Herkunft     :  STRING [50];
    END;
  TypVerzeichnis    =  ARRAY [TypApfelsorte] OF
                        TypSortenInformation;
VAR
  Verzeichnis       :  TypVerzeichnis;
```

Das letzte Beispiel zeigt eine typische Anwendung einer Schachtelung von ARRAYs. Jede Fläche, die in Zeilen und Spalten gegliedert ist, läßt sich als Doppel-ARRAY auffassen. Das heißt, sie wird behandelt als eine Reihe von Zeilen, die jeweils Spaltenelemente darstellen, oder auch umgekehrt. Ein solches Doppel-ARRAY heißt auch Matrix.

Programmieren eines Font-Editors

Alle handelsüblichen Matrixdrucker bieten die Möglichkeit, Zeichensätze zu generieren oder die Form nicht benötigter Zeichen umzuprogrammieren. Wenn der Drukker das Zeichen Ω, ASCII 234, empfängt, könnte er zum Beispiel ein kleines Telefonsymbol zu Papier bringen. Die neue Form des Zeichens muß allerdings punktweise in der Feinheit, die den Nadeln des Druckkopfs entspricht, definiert werden. Details hierzu sind im Handbuch des Druckers nachzulesen.

Die punktweise Beschreibung und Veränderung eines Zeichens soll ein Programm erleichtern, mit dem Zeichen

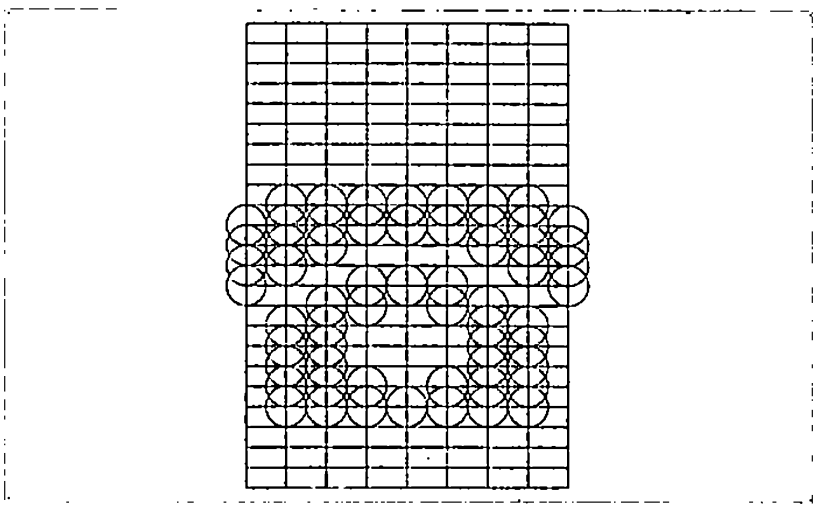

Abbildung 14-10: Symbol Telefon

des Druckzeichensatzes (des Fonts) in Blockgrafik am Bildschirm ediert werden können. Aus der dabei gewonnenen Beschreibung lassen sich nach der Methode, die in Ihrem Druckerhandbuch für Ihr Gerät angegeben ist, relativ einfach die Steuercodes erzeugen, die das neue Zeichen für den Drucker gültig machen.

Ähnlich können Sie sich ein Logo, Ihr individuelles grafisches Symbol, definieren (sozusagen ein *Stempel* in verfeinerter Ausführung und mit weit höherer Auflösung, als es beim allerersten Programm der Fall war).

Als hauptsächliche Datenstruktur wird ein Doppel-ARRAY, die Druckmatrix, benötigt. Dieses hält fest, welche Punkte (Druckernadeln) gesetzt und welche nicht gesetzt sind. Wenn nachher der Steuercode für den Drucker ermittelt wird, müssen lediglich die Komponenten der Druckmatrix (vom *TypDruckspalte*) mit einer bestimmten Formel ausgewertet werden. Ein Problem ist die halb-

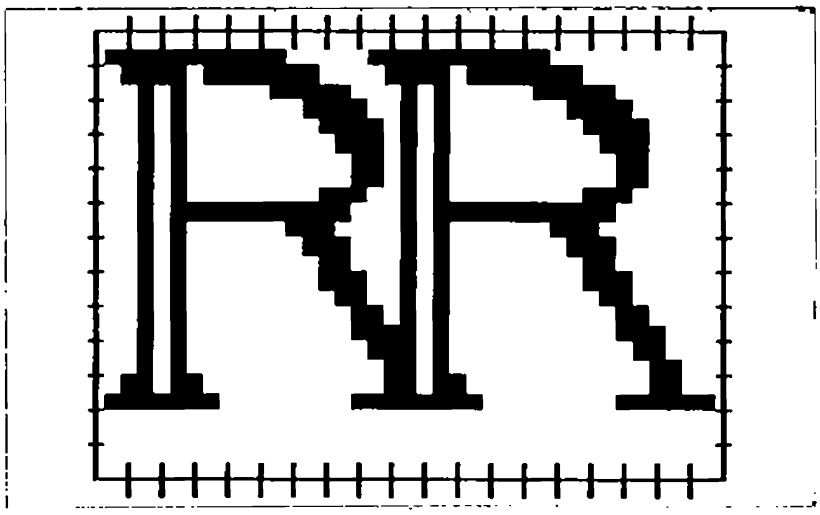

Abbildung 14-11: Logo

wegs korrekte Darstellung des momentanen Inhalts der Druckmatrix auf dem (nicht-grafischen) Bildschirm. Die normalen Zeichen des Bildschirms eignen sich dazu nur schlecht, weil sie die Abbildung extrem vertikal verzerren würden. Sie können sich aber beim PC-Zeichensatz mit den Zeichen '■' (ASCII 219), '▄' (ASCII 220) und '▀' (ASCII 223) behelfen. Der Trick besteht darin, jeweils zwei Druckmatrix-Punkte in einem Bildmatrix-Element, also einem Bildschirmzeichen, zu vereinen. '■' muß stehen, wenn die beiden Druck-Punkte gesetzt sind, '▄', wenn nur der untere Punkt, '▀', falls nur der obere Punkt verwendet werden soll. Als grundlegende Datenstruktur können Sie das auf der folgenden Seite beginnende Beispiel verwenden.

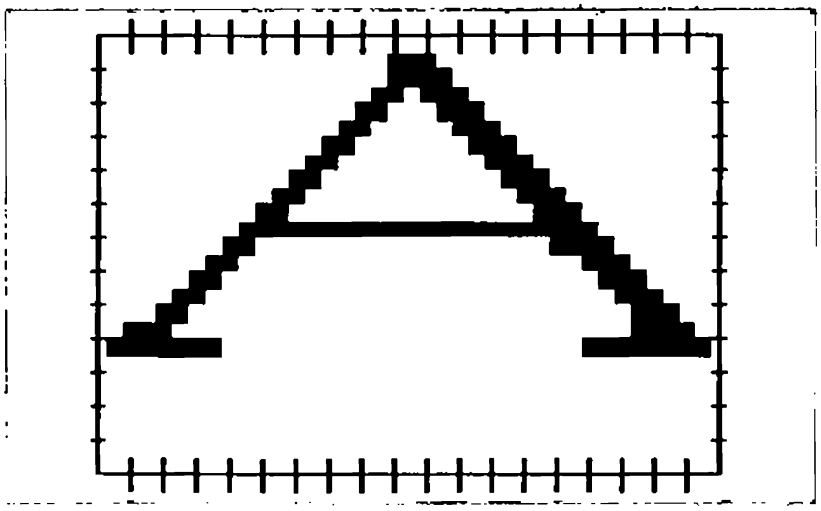

Abbildung 14-12: Ein „A"

```
CONST
                                  {Konstanten für die ARRAY-Grenzen,
                                  hier für 24-Nadeldrucker gewählt.}
  Horizontalaufloesung = 37;  {Anzahl Spalten}
  Vertikalaufloesung   = 24;  {Anzahl von Punkten je Spalte}
  MaxBildbreite        = 40;  {max. Anzahl Bildschirmspalten}
  MaxBildhoehe         = 20;  {maximale Zahl Bildschirmzeilen}
{Praktische Bezeichnungen}
  PunktGesetzt         = true;
  NichtGesetzt         = false;
  MenueAnfangszeile    = 2;
  MenueAnfangsspalte   = 1;
  BildAnfangszeile     = 5;
  BildAnfangsspalte    = 20;
TYPE
  TypPunktgesetzt      = Boolean;
  TypDruckspalte       = ARRAY [1..Vertikalaufloesung] OF
                         TypPunktgesetzt;
  TypDruckmatrix       = ARRAY [1..Horizontalaufloesung] OF
                         TypDruckspalte;
  TypBildzeile         = ARRAY [1..MaxBildbreite] OF Char;
  TypBildmatrix        = ARRAY [1..MaxBildhoehe]  OF TypBildzeile;
VAR
  Druckmatrix          : TypDruckmatrix;
  Bildmatrix           : TypBildmatrix;
                         {Die aktuellen Positionen in der
                         Druck- beziehungsweise der Bildmatrix}
  Druzeile, Druspalte  : Integer;
  Bilzeile, Bilspalte  : Integer;
```

Das weitere sei nun Ihnen überlassen.

Aufgaben

1. Skizzieren Sie Syntaxdiagramme der RECORD-, der ARRAY- und der FILE-Deklaration.

2. Die Weinkellerei Zucker & Schwefel will ihre wöchentliche Umsatzstatistik von der EDV aufbereiten lassen. Nach jeder Woche soll ein Bericht der folgenden Form gedruckt werden (s. S. 263, Abb. 14-13):

Stellen Sie eine Datenstruktur auf, die derartige Daten aufnehmen kann (wobei die Umsätze keine zu speichernde Größe sind). Schreiben Sie ein Programm, mit dem die

Sorte	Bezeichnung	Anzahl	Preis	Umsatz
1	Riesling	15	3.90	....
2	Müller-Thurgau	10	4.50	....
.....				

Abbildung 14-13: Wochenbericht der Weinkellerei Zucker & Schwefel

Daten und Rechenergebnisse aufgenommen und ausgegeben werden können.

3. Ein ARRAY ['a'..'z'] OF Integer besteht aus so vielen Integer-Objekten, wie es Buchstaben im Alphabet gibt, und ergibt die ideale Datenstruktur für ein Programm, das die Häufigkeit der verschiedenen Buchstaben des Alphabets in einem Text zählt. Schreiben Sie ein solches Programm. Gehen Sie von Texten ohne deutsche Sonderzeichen aus. Versuchen Sie später, das Programm so abzuändern, daß auch diese berücksichtigt werden.

4. Das Schachbrett ist wirklich ein zweidimensionales ARRAY. Eine Datenstruktur, mit der sich Stellungen des Spiels abbilden lassen, wird beschrieben durch:

```
TYPE
  TypFigur        =
    (WBauer1, WBauer2, WBauer3, WBauer4,
     WBauer5, WBauer6, WBauer7, WBauer8,
     WTurm1, WTurm2, WRoessel1, WRoessel2,
     WLaeufer1, WLaeufer2, WDame, WKoenig,
     SBauer1, SBauer2, SBauer3, SBauer4,
     SBauer5, SBauer6, SBauer7, SBauer8,
     STurm1, STurm2, SRoessel1, SRoessel2,
     SLaeufer1, SLaeufer2, SDame, SKoenig,
     Leer);
  TypSchachbrett : ARRAY ['a'..'h'] OF ARRAY['1'..'8'] OF TypFigur;
VAR
  Schachbrett     : TypSchachbrett;
```

Zum Beispiel steht auf dem Feld b7 die Figur:

```
Schachbrett ['b'] [7]
```

Erweitern Sie diese Datenstruktur so, daß Spielfolgen (oder ganze Partien) dargestellt werden können. Definie-

ren Sie eine Prozedur *Zug*, mit der ein Spielzug protokolliert werden kann.

7. Wie könnte eine Datenstruktur aussehen, mit der sich die Spielstellung von Mensch-Ärgere-Dich-nicht oder Malefiz beschreiben ließe? Eine etwas vereinfachte Form der Spiele läßt sich mit gar nicht so großem Aufwand als Computerspiel programmieren.

8. Wenn Sie Interesse an Statistik haben, können Sie dank der ARRAYs jetzt deren Methoden einsetzen. Ein ARRAY [1..N] of Real mit einer Konstante N, etwa N = 30, speichert eine Zahlenreihe in der Form, wie sie für die statistische Auswertung nach den üblichen Formeln benötigt wird.

9. Programmieren Sie einen Auskunftsautomaten für den Geldtausch vor dem Urlaub.

- ❒ Für die wichtigsten Urlaubsländer werden die Wechselkurse erfaßt
- ❒ und als Kurstafel in einer Übersicht angezeigt.

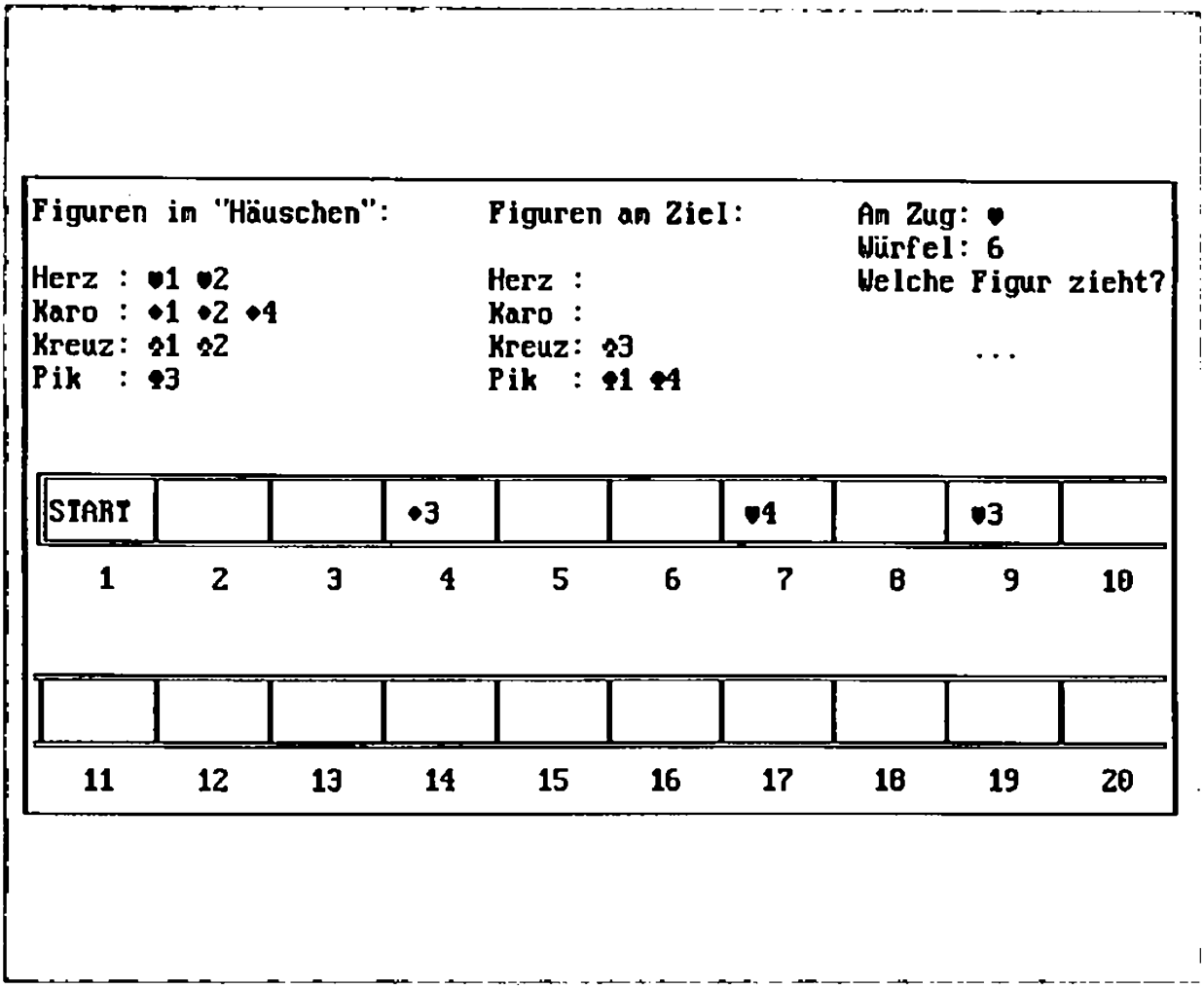

Abbildung 14-14: „Mensch-Ärgere-Dich-Nicht"

- Auf Anforderung wird berechnet, welcher Betrag einer ausgesuchten ausländischen Währung einem bestimmten DM-Betrag entspricht.

  Legen Sie dabei die folgende oder eine ähnliche Datenstruktur zugrunde:

```
TYPE
  TypWaehrungen =
     (Dollar, Franc, Pfund, Lira, Franken, Schilling, Peseten,
      Dinar, Forint, Rubel);
  TypKurstafel  = Array [TypWaehrungen] of Real;
VAR
  Kurstafel     : TypKurstafel;
```

Anhang: Der IBM-Zeichensatz

								80	50	01010000	P
								81	51	01010001	Q
								82	52	01010010	R
								83	53	01010011	S
0	00	00000000		40	28	00101000	(	84	54	01010100	T
1	01	00000001	☺	41	29	00101001	)	85	55	01010101	U
2	02	00000010	☻	42	2A	00101010	*	86	56	01010110	V
3	03	00000011	♥	43	2B	00101011	+	87	57	01010111	W
4	04	00000100	♦	44	2C	00101100	,	88	58	01011000	X
5	05	00000101	♣	45	2D	00101101	–	89	59	01011001	Y
6	06	00000110	♠	46	2E	00101110	.	90	5A	01011010	Z
7	07	00000111	•	47	2F	00101111	/	91	5B	01011011	[
8	08	00001000	◘	48	30	00110000	0	92	5C	01011100	\
9	09	00001001	○	49	31	00110001	1	93	5D	01011101	]
10	0A	00001010	◙	50	32	00110010	2	94	5E	01011110	^
11	0B	00001011	♂	51	33	00110011	3	95	5F	01011111	_
12	0C	00001100	♀	52	34	00110100	4	96	60	01100000	`
13	0D	00001101	♪	53	35	00110101	5	97	61	01100001	a
14	0E	00001110	♫	54	36	00110110	6	98	62	01100010	b
15	0F	00001111	☼	55	37	00110111	7	99	63	01100011	c
16	10	00010000	►	56	38	00111000	8	100	64	01100100	d
17	11	00010001	◄	57	39	00111001	9	101	65	01100101	e
18	12	00010010	↕	58	3A	00111010	:	102	66	01100110	f
19	13	00010011	‼	59	3B	00111011	;	103	67	01100111	g
20	14	00010100	¶	60	3C	00111100	<	104	68	01101000	h
21	15	00010101	§	61	3D	00111101	=	105	69	01101001	i
22	16	00010110	▬	62	3E	00111110	>	106	6A	01101010	j
23	17	00010111	↨	63	3F	00111111	?	107	6B	01101011	k
24	18	00011000	↑	64	40	01000000	@	108	6C	01101100	l
25	19	00011001	↓	65	41	01000001	A	109	6D	01101101	m
26	1A	00011010	→	66	42	01000010	B	110	6E	01101110	n
27	1B	00011011	←	67	43	01000011	C	111	6F	01101111	o
28	1C	00011100	∟	68	44	01000100	D	112	70	01110000	p
29	1D	00011101	↔	69	45	01000101	E	113	71	01110001	q
30	1E	00011110	▲	70	46	01000110	F	114	72	01110010	r
31	1F	00011111	▼	71	47	01000111	G	115	73	01110011	s
32	20	00100000		72	48	01001000	H	116	74	01110100	t
33	21	00100001	!	73	49	01001001	I	117	75	01110101	u
34	32	00100010	"	74	4A	01001010	J	118	76	01110110	v
35	23	00100011	#	75	4B	01001011	K	119	77	01110111	w
36	24	00100100	$	76	4C	01001100	L	120	78	01111000	x
37	25	00100101	%	77	4D	01001101	M	121	79	01111001	y
38	26	00100110	&	78	4E	01001110	N	122	7A	01111010	z
39	27	00100111	'	79	4F	01001111	O	123	7B	01111011	{

1. Spalte: Dezimal; 2. Spalte: Hexadezimal; 3. Spalte: Binär; 4. Spalte: Zeichen

124	7C	01111100	¦
125	7D	01111101	}
126	7E	01111110	~
127	7F	01111111	⌂
128	80	10000000	Ç
129	81	10000001	ü
130	82	10000010	é
131	83	10000011	â
132	84	10000100	ä
133	85	10000101	à
134	86	10000110	å
135	87	10000111	ç
136	88	10001000	ê
137	89	10001001	ë
138	8A	10001010	è
139	8B	10001011	ï
140	8C	10001100	î
141	8D	10001101	ì
142	8E	10001110	Ä
143	8F	10001111	Å
144	90	10010000	É
145	91	10010001	æ
146	92	10010010	Æ
147	93	10010011	ô
148	94	10010100	ö
149	95	10010101	ò
150	96	10010110	û
151	97	10010111	ù
152	98	10011000	ÿ
153	99	10011001	Ö
154	9A	10011010	Ü
155	9B	10011011	¢
156	9C	10011100	£
157	9D	10011101	¥
158	9E	10011110	₧
159	9F	10011111	ƒ
160	A0	10100000	á
161	A1	10100001	í
162	A2	10100010	ó
163	A3	10100011	ú
164	A4	10100100	ñ
165	A5	10100101	Ñ
166	A6	10100110	ª
167	A7	10100111	º
168	A8	10101000	¿
169	A9	10101001	⌐
170	AA	10101010	¬
171	AB	11101011	½
172	AC	10101100	¼
173	AD	10101101	¡
174	AE	10101110	«
175	AF	10101111	»
176	B0	10110000	░
177	B1	10110001	▒
178	B2	10110010	▓
179	B3	10110011	│
180	B4	10110100	┤
181	B5	10110101	╡
182	B6	10110110	╢
183	B7	10110111	╖
184	B8	10111000	╕
185	B9	10111001	╣
186	BA	10111010	║
187	BB	10111011	╗
188	BC	10111100	╝
189	BD	10111101	╜
190	BE	10111110	╛
191	BF	10111111	┐
192	C0	11000000	└
193	C1	11000001	┴
194	C2	11000010	┬
195	C3	11000011	├
196	C4	11000100	─
197	C5	11000101	┼
198	C6	11000110	╞
199	C7	11000111	╟
200	C8	11001000	╚
201	C9	11001001	╔
202	CA	11001010	╩
203	CB	11001011	╦
204	CC	11001100	╠
205	CD	11001101	═
206	CE	11001110	╬
207	CF	11001111	╧
208	D0	11010000	╨
209	D1	11010001	╤
210	D2	11010010	╥
211	D3	11010011	╙
212	D4	11010100	╘
213	D5	11010101	╒
214	D6	11010110	╓
215	D7	11010111	╫
216	D8	11011000	╪
217	D9	11011001	┘
218	DA	11011010	┌
219	DB	11011011	█
220	DC	11011100	▄
221	DD	11011101	▌
222	DE	11011110	▐
223	DF	11011111	▀
224	E0	11100000	α
225	E1	11100001	ß
226	E2	11100010	Γ
227	E3	11100011	π
228	E4	11100100	Σ
229	E5	11100101	σ
230	E6	11100110	µ
231	E7	11100111	τ
232	E8	11101000	Φ
233	E9	11101001	Θ
234	EA	11101010	Ω
235	EB	11101011	δ
236	EC	11101100	∞
237	ED	11101101	φ
238	EE	11101110	ε
239	EF	11101111	∩
240	F0	11110000	≡
241	F1	11110001	±
242	F2	11110010	≥
243	F3	11110011	≤
244	F4	11110100	⌠
245	F5	11110101	⌡
246	F6	11110110	÷
247	F7	11110111	≈
248	F8	11111000	°
249	F9	11111001	∙
250	FA	11111010	·
251	FB	11111011	√
252	FC	11111100	ⁿ
253	FD	11111101	²
254	FE	11111110	■
255	FF	11111111	

1. Spalte: Dezimal; 2. Spalte: Hexadezimal; 3. Spalte: Binär; 4. Spalte: Zeichen

Register

16-Bit-Code 21
24-Nadeldrucker 262
3-Bit-Code 12
8-Bit-Code 12

Abbruchbedingungen 126
abs 225
Absatz 149, 152
Absolutbetrag 225
ABSOLUTE 27
Abstraktion 26
Absturz 194
Addition 83, 192
Aktualparameter 212, 218
Alphabet 28, 67
Alt-Taste 31
AND 27, 104
Anfangsspalte 42, 45
Anfangswert 59, 62, 137
Anfangszeile 42, 45
Anweisungen 28
  Folge 45
  Handlungs- 25
Anweisungsteil 29, 34, 49
Apostroph 43, 55
Arbeitsdatei 18
Arbeitslaufwerk 18
ARRAY 27, 233, 242, 245–247, 252, 259
ARRAY OF
  Real 247
ARRAY-Grenzen 247
ARRAY-Komponente 243
ARRAYs 242, 244
  Schachtelung 259
ASCII 12, 31, 92–94, 161, 180, 200
  Codenummer 144
  Datei 14
ASCII-Tabelle 43, 256
ASM 27
ASSEMBLER 27
Assign 142–144, 147–148, 160, 239, 245, 254
Attribute 235
Aufzählungstyp 172, 184
Aufzählungstypen 259
  selbstdefinierte 189–190, 192
Ausgabe 190
Ausgabeformat 52
Ausgabegrößen 88
ausgelagert
  in Dateien 253
Ausschnittstypen 184, 192
Auswahlstruktur 99
AUX 148

Balkendiagramm 98
Baustein 32, 217, 219
Bausteine 39, 214, 218–219, 221
Bedingungen 100, 106
  Entscheidungs- 113
  geschachtelte 111
  Wiederholungs- 113
Bedingungsausdruck 125
Bedingungsausdrücke 103
Befehlssatz 11
BEGIN 27–28, 34, 102
BEGIN-END 62, 173
BEGIN-END-Rahmen 65, 105, 120, 238
beschrieben
  explizit 237
  parametrisch 237
Betriebssystem 141, 257
Bildmatrix 262
Bildschirm 140
  ausgeben auf 43
Bildschirmausgabe 35, 51
Bildschirmmaske 16, 73
Bildzeile 262
Bit 11, 50, 52
Bitmuster 11
Blaise Pascal 14
Boolean 52, 62, 104, 125, 159, 183
boolesche Schleifenkontrollgröße 128
boolescher Wert 143
Bottom-up 39, 215
Bottom-up-Verfahren 40
Buchstaben 28
Byte 50, 193
Bytes 11, 53

C(ompile) 26
Cardinal 193, 200, 202, 212
CASE 27, 252
CASE-Anweisung 170, 172–173, 179–180, 191
Change Direktory 18
Char 52, 62, 92, 105, 153, 172, 183, 244
Char-Komponenten 248
Char-Werte 93, 168, 225
Chars 153
Chr 225
chr-Funktion 92
Close 143–144, 160, 239, 245, 254
ClrScr 35–36
Code 11
COM 148
Compiler 14, 33, 35
Compilieren 33
Computerwörter 11
CON 148
Concat 153
CONST 27, 37, 42, 48, 244
CONSTRUCTOR 27
Copy 154, 158, 220
Cursor 78, 178, 219

Datei 17, 144
  Öffnen 142
  ordnen 257
Datei mit Zeilenstruktur 150
Dateien 139–141
  Öffnen 239, 240
  Schließen 143, 239
  strukturierte 254
Dateiname 17
Dateivariable 140–141
Dateizeiger 255
Daten 22
  Interpretation 23
Datenanpassungen 148
Datenelemente 235
Datenerfassung 75
Datenexport 211
Datenfeld 235, 256
Datenfeldnamen 235
Datenimport 211
Datensatz 234–235
Datensatz-Typ 233
Datenstrukturierung 250
Datentyp 54
  komplex 184

strukturiert 184
zusammengesetzt 184
Datentypen 50, 57
Kreation 183
Datenverbund 233
Debugger 15
Deklaration 228, 234
explizite 243
Deklarationsteil 29, 32, 34, 37, 41, 48, 53, 161
Delay 60, 179, 203
Delay-Anweisung 69
Delete 154
DESTRUCTOR 27
Dezimalsystem 12
Dezimalzahl 50, 89, 155, 225
Dialogprogramm 80
DIN-Diagramm 101
DIN-Norm 22
66001 107
Directory 17–18
Direktzugriff 253, 255
Disketten
Lesen 139
Schreiben 139
Diskettenlaufwerk 17
DIV 27, 86–87, 91, 96
Dividieren 84, 162
Division 83, 85
mit Rest 85
DO 27, 61–62, 65, 120
Doppel-ARRAY 259
Doppelpunkt 48–49, 51, 173
DOWNTO 27, 66
Drucker 36, 56, 78–79, 260
Druckeranpassung 179
Druckmatrix 260, 262
Druckspalte 262
Dualsystem 12
Dummy-Prozeduren 168, 199
Durchschnittswert 146

E-V-A-Schema 76, 88
Ebenen 18
Edit 18
Editor 18, 39, 144, 156
Turbo 19
Eingabe 72, 190
-dialog 72
Eingabeanweisung 71, 237
Eingabegrößen 88
Eingabekontrolle 178, 229
Eingabemaske 74
Einruecken 79
Einzelanweisungen 41
Elementtyp 197
ELSE 27, 101, 105–106, 110–111
END 27–28, 32, 34, 102, 172, 234
Endwert 59, 62, 137
ENTER-Taste 59
Entscheidungen
einseitige 110
Entscheidungsstruktur 99, 108–109, 126, 251
Entwicklungsumgebung 19
integrierte 15–16
EOF 143–144, 159–160, 177
EoLn 152, 161, 225
Escape-Taste 18
explizite Typen 234
EXTERNAL 27

Fallunterscheidung
flache 166
falsch 104
false 52, 104, 167
FAR 27
Farbbildschirmen 164
Fehleingaben 167
Korrektur 168
Fehleranzeige
akustische 175
Fehlermeldungen 15, 72, 175, 179
Festplatte 15
FILE 27, 233, 254
FILE OF 253
flache Wiederholungsstruktur 231
Flag 125
Fließtext 148–149, 156
Font-Editor 260
Fonts 260
FOR 27, 60, 202, 243
For-Anweisung 57–58, 60–66, 68, 190–191
FOR-Schleife 59, 243
Formalparameter 212
Formular 73
FORWARD 27, 228
FUNCTION 27, 226–227
Funktion 227
boolesche 227
Funktionen 154, 192, 216, 224, 228
selbstdefinierte 226
Funktionsdeklaration 226

Ganzzahlen 89
Geräte 139
Get-/Put-Operationen 254
Gleichheitszeichen 49
Gleitkommazahl 50
Gleitpunktdarstellung 50
Gleitpunktzahl 50
Global 208–209
globale Größen 208
globale Variable 208, 211
globale Variablen 208
GOTO 27, 117
GoToXY 35–36, 39, 75, 78, 117
Grobgliederung 29
Großbuchstaben 171
Großschreibung 28
Grundtyp 197

Hardcopy 78
Hauptmenü 16, 17
Hauptprogramm 48
Helligkeiten 164
Hexadezimalsystem 12
Hilfsvariable 250
Horizontalaufloesung 262

IBM-Zeichensatz 43–44, 92, 266
IF 27, 101
IF-Anweisung 102, 105–106, 110–111, 152
IF-THEN-ELSE 102, 164, 174
IF-THEN-ELSE-Anweisungen 101, 113
geschachtelt 166
IF-THEN-ELSE-Schachtelung 167
IMPLEMENTATION 27
Implementationen 50
Pascal- 141
implizite 234
implizite Typvereinbarung 234
IN 27
Include 216
Index 243, 258
Indexgrenze 244
Information 22

INLINE 27
Input 27
Insert 155
Integer 49–50, 88, 90, 92, 172, 183
Integer-Variable 155, 191, 193
Integer-Wert 91, 153
INTERFACE 27
INTERRUPT 27

JA-Programmast 107
JA-/NEIN-Teil 106
JA-Zweig 101
Jackson-Diagramme 250

Kardinalzahlen 193
kbd 135, 159, 170
KeyPressed 225
Klammerart 33
Klammeraffe 28
Klammern 33, 55, 83, 205
Kleinbuchstaben 171, 196
Kleinschreibung 28
Kommentare 33, 34, 45
komplexe Datenstrukturen 233
Komponente 235
Komponentennamen 238
Konstanten 31, 38–39, 42–43, 46, 48–49, 103, 131
-definition 42
-deklaration 37
Char- 52
Integer- 51
Textzeichen 57
Zahlen- 82
Kontrollvariable 61
Koordinaten 35
Kopierfunktion 39
Korrekturmöglichkeit 167

LABEL 27
LABEL-Deklarationen 187
Laufschleife 59, 61
Laufwerk 18
Leerstrings 79
Leerzeichen 61
Leerzeilen 149
Length 74, 77, 178, 220
Length ( ) 67
Lesen
zeichenweise 144
Load 18
lock 143
Logik 13, 104, 111, 125
Logo 261
lokal 210
lokale Größen 208
lokale Variable 208, 211, 213
LongInt-Werte 89
LPT 148
LST 79, 140, 148

Mantisse 50
Maschinensprache 11–13
Masken 242
-prozedur 75
Matrix 259
Maus 164
Maximallängen 234
MaxInt 200
Mehrfachauswahl 164
Menü-Oberfläche 163
Menüleiste 16, 170
Menüpunkte 164
Menüs 168, 242
Mit 192
Mittelwertsfunktion 227
MOD 27, 86–87, 91, 96
MOD-Operation 92
Modul 214
Modula2 214
Modularisierung 213, 215
Module 216
Modulkonzept 214
Multiplikation 83
Multiplikationszeichen 85

nach oben schieben
Schirm 160
Nachkommastellen 52
Namen 28, 233
Nassi-Shneiderman-Diagramme 109
NEAR 27
NEIN-Programmast 107
NEIN-Zweig 101
Niklaus Wirth 9, 14
NIL 27
NOT 27, 104, 125
Null-Fehlersyndrom 177
Nummernzeichen # 43

Oberon 214
OBJECT 27
Objekte
komplexe 235, 257
Objektstruktur 233
OF 27, 259
Optionsaufrufen 169
OR 27, 125
ord 94, 192, 225
ord-Funktion 93, 180
Output 27

PACKED 27
Palindrome 70
PAP 107–108, 110, 115, 118–119
Parameter 205, 206–207, 212
typgebunden 248
parametrisieren 213
parametrisierte Prozeduren 216
Pascal
Standard 27
Pascal-Compiler 9, 14
Pascal-Dialekt 141
Pascal-Dialekte 27, 143
Pascal-Name 64–65, 235
Pascal-Programm 29, 64
Pascal-Programmkopf 64
Pascal-Wörter 27
Pos 152–153
pred 192, 225
printer 36
PRIVATE 27
PRN 148
Problemaufbaugraph 250
PROCEDURE 27, 32, 43
Produkt 85
PROGRAM 27, 32
Programm 13
-name 27
Programmabbruch 117
Programmablaufphase 107
Programmabsturz 100, 230
Programme
Anpassung 39
Strukturierung 41
Programmeldungen 164
Programmiermethodik 88
Programmiersprache 14, 25, 63
maschinenorientierte 13
problemorientierte 13
Programmiersprachen 51
4. Generation 242
Programmierstil 37

Programmierung
objektorientierte 14
Programmkopf 29, 65
Programmstruktur
Sequentielle 100
Prozedur 43, 45, 62, 76, 122, 155, 206–207, 213, 218, 220
-definition 32
-länge 110
Aufruf 32
Zeichnen 249
Prozeduraufruf 230
Prozedurdeklaration 226
Prozeduren 31–32, 41, 48, 113, 154, 205, 216, 228
Aufnehmen 242
Ausgabe- 248
Bearbeitungs- 248
definieren 248
geschachtelte 228
Kontroll- 248
Teil- 40
Wiedergeben 242
Prozedurkopf 34, 206–207
Prozedurparameter 235
Prozessor 11, 24
PrtSc 39
Puffer 142
Pull-down-Menü 18
Punkt-/Strichrechnung 85
Punktrechnung 83

Quadratwurzel 226
Quadrieren 226
Quelldatei 33
Quelle 139
Quick-Pascal 9

R(un) 26
Rahmen 34
Random 127–129, 202
Randomize 128
Read 71–72, 144, 152, 190, 254
ReadKey 93–94, 135, 178–179
ReadLn 59, 71, 144
Real 50, 84, 88–90, 92, 172, 183
Real-Variable 155, 191
Real-Variablen 62
Real-Wert 91, 225
Real-Werte 198
reale Objekte 233
Rechenoperation 172
Rechnen 82
RECORD 27, 233–234, 242, 244–245, 247, 249, 251, 253
Definition 252
RECORD-Deklaration 234
RECORD-Komponenten 248
Referenzzeile 16
Rekursion 229–230
rekursive Methode 230
REPEAT 27
REPEAT-UNTIL 175–177
Reset 142, 144, 160, 221, 240, 254
Return 20, 149
RETURN-Taste 59, 72
ReWrite 142, 239, 245, 254
round 90, 225
Rundung 89, 162
Rundungsfehler 83

Scancode 93–94
Schachbrett 263
Schachtelung 106, 112
flache 166
tiefe 166
Schleife 59, 92, 116–117, 222
Anfangszustand 123
Endzustand 124
Finalzustand 123
Initialzustand 123
Summenbildung 130
Schleifenbedingung 125
Schleifeninitialisierung 130
Schleifenkontrollvariable 167
Schleifenvariable 59
Schlüsselwörter 27–28, 32
Schnittstelle 214
Schrägstrich 85
Schreib-/Lesefenster 254
schreiben, groß, klein 27
Seek 255
Seiteneffekte 209
selbstdefinierter Aufzählungstyp
234
Semikolon 32
Senke 139
Sequentielle Struktur 109
Sequenz 99, 116
SET 27, 198
Deklaration 196
SET OF 197, 199, 217
SET-Typ, 195
strukturiert 197
SET-Variable 196
SHL 27
SHR 27
Signalvariable 125
Silbentrennprogramm 156
Silbentrennung 159
Simulation 127, 133, 136
Sonderzeichen 161
deutsche 28
Sortieralgorithmus 255
Sortieren 255, 257
alphabetisch 242, 255–256
alphanumerisch 255
nach Größe 256
Spalte 219
Speicher 45
externe 255
Zugriff 254
Speichern 239, 242
Sprachen
Lehrsprache 14
problemorientierte 14
sqr 225
sqrt (a) 226
Standard-Pascal 37, 48, 50, 54
Standardprozeduren 242
Standardtypen 183
Stempel 25–26, 31–32, 41
Steueranweisungen 35
Steuercode 260
Steuervariable 61
Steuerzeichen 93
Strichrechnung 83
STRING 27, 54, 69–70, 152–155, 165, 172, 198, 220
-Wert 94
auflösen 68
Umwandlung 95
STRING-Konstante 175
STRING-Typen 234
STRING-Variablen 55, 62, 67, 143
STRING-Werte 219
STRINGs 94, 149, 244
Struktogramm 109–110, 112, 115, 119, 122, 137–138, 177
Strukturdiagramm 44
Strukturen
geschachtelte 110
Strukturierung
hierarchische 41
Subtraktion 84, 192, 202

succ 192, 202, 225
suchen 242
Summe 85
Syntax 63
-definitionen 27
Syntaxdiagramm 63–65
Systemabsturz 229

Tastatur 24, 29, 140
Taste 94
Taste F1–Help 19
Teilbereichstypen 184, 192, 194
Text-Variablen
Deklaration 141
Textdatei 143, 239, 253–254
Texte
linksbündig 220
Textstücke 42, 55
Textverarbeitung 156
Textverarbeitungsprogramm 14, 19
THEN 27, 101
Tilde 28
TO 27, 58, 60, 66
Top-down 215
Top-down-Methode 40–41
true 52, 104
trunc 90–91, 225
Turbo-Hauptmenü 17
Turbo-Pascal 78
Typ 48, 92, 184–185
Typ-Inkompatibilität 186
Typdefinitionen 184, 188
explizite 194
implizite 194
Typdeklaration 185–186, 188
implizite 206
Typdeklarationsabschnitte 217
TYPE 27, 188–189
Typen 50, 52, 90, 186
Aufzählungs- 62
neue 184
Typvereinbarung 217, 240
UNIT 27, 36
Units 214, 222
Unterfunktionen 228
Untermenüs 164
Unterprozeduren 48, 228
UNTIL 27
USES 27, 78, 170, 187
USES crt 36

Val 155
VAR 27, 48–49, 53, 141, 188
Variable
Boolsche 52, 167
Char- 52
Variablen 45–46, 49, 51, 103, 184–186
äußere 76
Deklaration 48
prozedurinterne 76
RECORD- 236–237
STRING 52
Variablennamen 238
Variablenparameter 210–212, 218, 222, 226
Variablentypen
einfache 50
Verbundanweisung 65, 173
Vereinbarungsteil 29
Verfeinerung
schrittweise 40
Vergleich 103
Vergleichsoperatoren 103
Verketten
Chars 153
STRINGs 153
Verknüpfungen
logische 104
Version 3.x 95
Vertikalaufloesung 262
Verzögerungsanweisung 60
Verzweigung 108
VIRTUAL 27
Vorzeichen 50

Wagenrücklauf 19
wahr 104
Werkzeuge (Tools) 216
Wert 49, 184–186
Werteliste 180, 197
Werteparameter 210–212, 218–219, 226
Wertzuweisung 49, 67, 103
Wertzuweisungen 189
WHILE 27, 118, 124, 126, 177, 202
WHILE-Anweisung 118, 120, 124
Wiederholungsstruktur 114, 116, 119, 126, 229
WITH 27, 237, 245
Wordstar
Kompatibilität 19
Wortgrenzen 159
Write 28, 35, 67, 78, 190, 254
Write/WriteLn 42
WriteLn 28, 35, 150

XOR 27

Zahlen
ganze 49, 84
Zählschleifen 59, 60
Zeichen 24, 42
Zeichenketten 54, 67
Zeichensatzgrafik 98
Zeichensatztabelle 52
Zeile 219
Zeilenabschluß 29
zeilenorientiert 148
Zeilenstruktur 144
Zeilenumbruch 20
Zentrieren 218
Zufallsgenerator 128
Zugriffspfad 18

Computer verständlich

Einführung in Pascal

Die Programmiersprache Pascal hat BASIC von der Verbreitung her überholt und ist zur wichtigsten Programmiersprache geworden. Auch an Schulen und Hochschulen wird inzwischen hauptsächlich mit Pascal gearbeitet.
Dieses FALKEN Buch vermittelt dem Leser alle wichtigen Strukturen und Befehle der Programmiersprache Pascal. Es konfrontiert ihn nicht mit isolierten, abstrakten Beispielen, sondern erarbeitet mit ihm im Verlauf attraktiver Lektionen funktionsfähige Teilprogramme, die auch für das spätere Erstellen eigener Programme nützlich sind. Obwohl das Buch die Prinzipien des Programmierens mit Pascal erschließt und nicht auf bestimmte Pascal-Dialekte abhebt, wurden alle Beispielprogramme mit den neuesten Pascal-Versionen von Microsoft und Borland getestet.
Das Buch kann bei richtiger Nutzung traditionelle Schulungen und Seminare ersetzen, aber auch als Begleitmaterial zu diesen dienen.

Rupert Röder ist Diplom-Mathematiker und als Dozent für Informatik an einer privaten Computerschule tätig. Er leitet seit vielen Jahren Pascal-Kurse an der Volkshochschule in Mainz.

ISB N 3-8068-4329-5